Package 'lgr'

August 20, 2019

Type Package

Title A Fully Featured Logging Framework

Version 0.3.2

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, and databases, as well as email and push notifications. For a full list of features with examples please refer to the package vignette.

License MIT + file LICENSE

URL https://s-fleck.github.io/lgr

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

Depends R (>= 2.10)

Imports R6 (>= 2.4.0)

Suggests cli, covr, crayon, data.table, DBI, desc, future, future.apply, glue, gmailr, jsonlite, knitr, RMariaDB, rmarkdown, rotor (>= 0.2.2), RPostgres, rprojroot, RPushbullet, RSQLite, rsyslog, sendmailR, testthat, tibble, tools, utils, whoami, yaml

VignetteBuilder knitr

Encoding UTF-8

LazyData true

RoxygenNote 6.1.1

Collate 'Filterable.R' 'utils-sfmisc.R' 'utils.R' 'Appender.R' 'Filter.R' 'log_levels.R' 'print_LogEvent.R' 'Layout.R' 'LogEvent.R' 'Logger.R' 'basic_config.R' 'default_functions.R' 'get_logger.R' 'lgr-package.R' 'logger_config.R' 'logger_tree.R' 'print_Appender.R' 'print_Logger.R' 'read_json_log.R' 'simple_logging.R' 'use_logger.R' 'utils-formatting.R' 'utils-logging.R' 'utils-rd.R' 'utils-rotor.R' 'utils-test.R'

NeedsCompilation no

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

Repository CRAN

Date/Publication 2019-08-20 15:10:02 UTC

R topics documented:

AppenderBuffer *Log to a memory buffer*

Description

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

Usage

```
x <- 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 = "fatal", flush_on_exit = TRUE, flush_on_rotate = TRUE,
  should_flush = default_should_flush, filters = NULL)

x$add_appender(appender, name = NULL)
x$add_filter(filter, name = NULL)
x$append(event)
x$filter(event)
x$flush()
x$format(...)
x$format(color = FALSE, ...)
x$remove_appender(pos)
x$remove_filter(pos)
x$set_appenders(x)
x$set_buffer_size(x)
x$set_filters(filters)
x$set_flush_on_exit(x)
x$set_flush_on_rotate(x)
x$set_flush_threshold(level)
x$set_layout(layout)
x$set_should_flush(x)
x$set_threshold(level)
x$show(threshold = NA_integer_, n = 20L)

x$appenders
x$buffer_df
x$buffer_dt
x$buffer_events
```

```
x$buffer_size
x$data
x$destination
x$dt
x$filters
x$flush_on_exit
x$flush_on_rotate
x$flush_threshold
x$layout
x$should_flush
x$threshold
```

Creating a Buffer Appender

The Layout for this Appender is used only to format console output of its `$show()` method.

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.

`buffer_size`, `set_buffer_size(x)` `integer` scalar `>= 0` Number of LogEvents to buffer.

`buffer_events`, `buffer_df`, `buffer_dt` The contents of the buffer as a `list` of LogEvents, a `data.frame` or a `data.table`.

`flush_threshold`, `set_flush_threshold()` `integer` or `character` log level. Minimum event level that will trigger flushing of the buffer. This behaviour is implemented through `should_flush()`, and you can modify that function for different behaviour.

`should_flush(event)`, `set_should_flush(x)` A function with exactly one arguments: `event`. If the function returns `TRUE`, flushing of the buffer is triggered. Defaults to flushing if an event of level `error` or higher is registered.

`dt` Get the log recorded by this `Appender` as a `data.table` with a maximum of `buffer_size` rows

`data` Get the log recorded by this `Appender` as a `data.frame`

`threshold`, `set_threshold(level)` `character` or `integer` scalar. The minimum log level that triggers this logger. See log_levels

`layout`, `set_layout(layout)` a `Layout` that will be used for formatting the `LogEvents` passed to this Appender

`destination` The output destination of the `Appender` in human-readable form (mainly for print output)

`filters`, `set_filters(filters)` a `list` that may contain `functions` or any R object with a `filter()` method. These functions must have exactly one argument: `event` which will get passed the LogEvent when the Filterable's `filter()` method is invoked. If all of these functions evaluate to `TRUE` the LogEvent is passed on. Since LogEvents have reference semantics, filters can also be abused to modify them before they are passed on. Look at the source code of `with_log_level()` or `with_log_value()` for examples.

Methods

`flush()` Manually trigger flushing

`add_appender(appender, name = NULL)`, `remove_appender(pos)` Add or remove an Appender. Supplying a `name` is optional but recommended. After adding an Appender with `appender$add_appender(AppenderCo = "console")` you can refer to it via `appender$appenders$console`. `remove_appender()` can remove an Appender by position or name.

`flush()` Manually trigger flushing of the buffer

`show(n, threshold)` Show the last `n` log entries with a log level bellow `threshold`. The log entries will be formatted for console output via this Appenders Layout

`append(event)` Tell the Appender to process a LogEvent `event`. This method is usually not called by the user, but invoked by a Logger

`filter(event)` Determine whether the LogEvent `x` should be passed on to Appenders (`TRUE`) or not (`FALSE`). See also the active binding `filters`

`add_filter(filter, name = NULL)`, `remove_filter(pos)` Add or remove a filter. When adding a filter an optional `name` can be specified. `remove_filter()` can remove by position or name (if one was specified)

Comparison AppenderBuffer and AppenderDt

Both AppenderBuffer and AppenderDt do in memory buffering of events. AppenderBuffer retains a copies of the events it processes and has the ability to pass the buffered events on to other Appenders. AppenderDt converts the events to rows in a `data.table` and is a bit harder to configure. Used inside loops (several hundred iterations), AppenderDt has much less overhead than AppenderBuffer. For single logging calls and small loops, AppenderBuffer is more performant. This is related to how memory pre-allocation is handled by the appenders.

In short: Use AppenderDt if you want an in-memory log for interactive use, and AppenderBuffer if you actually want to buffer events

See Also

LayoutFormat

Other Appenders: `AppenderConsole`, `AppenderDbi`, `AppenderFileRotating`, `AppenderFile`, `AppenderGmail`, `AppenderJson`, `AppenderPushbullet`, `AppenderRjdbc`, `AppenderSendmail`, `AppenderSyslog`, `AppenderTable`, `Appender`

AppenderConsole *Log to the console*

Description

A simple Appender that outputs to the 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).

Usage

```
x <- 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)

x$add_filter(filter, name = NULL)
x$append(event)
x$filter(event)
x$format(color = FALSE, ...)
x$remove_filter(pos)
x$set_filters(filters)
x$set_layout(layout)
x$set_threshold(level)

x$destination
x$filters
x$layout
x$threshold
```

Creating a New Appender

New Appenders are instantiated with `<AppenderSubclass>$new()`. For the arguments to `new()` please refer to the section *Fields*. You can also modify those fields after the Appender has been created with setters in the form of `appender$set_<fieldname>(value)`

Fields

`threshold`, `set_threshold(level)` `character` or `integer` scalar. The minimum log level that triggers this logger. See log_levels

`layout`, `set_layout(layout)` a `Layout` that will be used for formatting the `LogEvents` passed to this Appender

`destination` The output destination of the `Appender` in human-readable form (mainly for print output)

`filters`, `set_filters(filters)` a `list` that may contain `functions` or any R object with a `filter()` method. These functions must have exactly one argument: `event` which will get

passed the LogEvent when the Filterable's `filter()` method is invoked. If all of these functions evaluate to `TRUE` the LogEvent is passed on. Since LogEvents have reference semantics, filters can also be abused to modify them before they are passed on. Look at the source code of `with_log_level()` or `with_log_value()` for examples.

Methods

`append(event)` Tell the Appender to process a LogEvent `event`. This method is usually not called by the user, but invoked by a Logger

`filter(event)` Determine whether the LogEvent `x` should be passed on to Appenders (`TRUE`) or not (`FALSE`). See also the active binding `filters`

`add_filter(filter, name = NULL)`, `remove_filter(pos)` Add or remove a filter. When adding a filter an optional `name` can be specified. `remove_filter()` can remove by position or name (if one was specified)

See Also

LayoutFormat

Other Appenders: `AppenderBuffer`, `AppenderDbi`, `AppenderFileRotating`, `AppenderFile`, `AppenderGmail`, `AppenderJson`, `AppenderPushbullet`, `AppenderRjdbc`, `AppenderSendmail`, `AppenderSyslog`, `AppenderTable`, `Appender`

Examples

```
# create a new logger with propagate = FALSE to prevent routing to the root
# logger. Please look at the section "Logger Hirarchies" 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
```

`AppenderDbi` *Log to databases via DBI*

Description

Log to a database table with any **DBI** compatible backend. Please be aware that AppenderDbi does *not* support case sensitive / quoted column names, and you advised to only use all-lowercase names for custom fields (see `...` argument of LogEvent). When appending to a database table all LogEvent values for which a column exists in the target table will be appended, all others are ignored.

Buffered Logging

AppenderDbi does not write directly to the database but to an in memory buffer. With the default settings, this buffer is written to the database whenever the buffer is full (`buffer_size`, default is 10 LogEvents), whenever a LogEvent with a level of `fatal` or `error` is encountered (`flush_threshold`) or when the Appender is garbage collected (`flush_on_exit`), i.e. when you close the R session or shortly after you remove the Appender object via `rm()`. If you want to disable buffering, just set `buffer_size` to `0`.

Usage

```
x <- AppenderDbi$new(conn, table, threshold = NA_integer_, layout =
  select_dbi_layout(conn, table), close_on_exit = TRUE, buffer_size = 10,
  flush_threshold = "error", flush_on_exit = TRUE, flush_on_rotate = TRUE,
  should_flush = default_should_flush, filters = NULL)

x$add_filter(filter, name = NULL)
x$append(event)
x$filter(event)
x$flush()
x$format(color = FALSE, ...)
x$remove_filter(pos)
x$set_buffer_size(x)
x$set_close_on_exit(x)
x$set_conn(conn)
x$set_filters(filters)
x$set_flush_on_exit(x)
x$set_flush_on_rotate(x)
x$set_flush_threshold(level)
x$set_layout(layout)
x$set_should_flush(x)
x$set_threshold(level)
x$show(threshold = NA_integer_, n = 20)
x$show(threshold = NA_integer_, n = 20L)

x$buffer_df
x$buffer_dt
x$buffer_events
x$buffer_size
x$close_on_exit
x$col_types
x$conn
x$data
x$destination
x$dt
x$filters
x$flush_on_exit
x$flush_on_rotate
x$flush_threshold
```

```
x$layout
x$should_flush
x$table
x$table_id
x$table_name
x$threshold
```

Creating a New Appender

An AppenderDbi is linked to a database table via its `table` argument. If the table does not exist it is created either when the Appender is first instantiated or (more likely) when the first LogEvent would be written to that table. Rather than to rely on this feature, it is recommended that you create the target log table first manually using an `SQL CREATE TABLE` statement as this is safer and more flexible. See also LayoutDbi.

New Appenders are instantiated with `<AppenderSubclass>$new()`. For the arguments to `new()` please refer to the section *Fields*. You can also modify those fields after the Appender has been created with setters in the form of `appender$set_<fieldname>(value)`

Fields

Note: `$data` and `show()` query the data from the remote database and might be slow for very large logs.

`close_on_exit`, `set_close_on_exit()` `TRUE` or `FALSE`. Close the Database connection when the Logger is removed?

`conn`, `set_conn(conn)` a DBI connection

`table` Name of the target database table

`buffer_size`, `set_buffer_size(x)` `integer` scalar `>= 0` Number of LogEvents to buffer.

`buffer_events`, `buffer_df`, `buffer_dt` The contents of the buffer as a `list` of LogEvents, a `data.frame` or a `data.table`.

`flush_threshold`, `set_flush_threshold()` `integer` or `character` log level. Minimum event level that will trigger flushing of the buffer. This behaviour is implemented through `should_flush()`, and you can modify that function for different behaviour.

`should_flush(event)`, `set_should_flush(x)` A function with exactly one arguments: `event`. If the function returns `TRUE`, flushing of the buffer is triggered. Defaults to flushing if an event of level `error` or higher is registered.

`dt` Get the log recorded by this `Appender` as a `data.table` with a maximum of `buffer_size` rows

`data` Get the log recorded by this `Appender` as a `data.frame`

`threshold`, `set_threshold(level)` `character` or `integer` scalar. The minimum log level that triggers this logger. See log_levels

`layout`, `set_layout(layout)` a `Layout` that will be used for formatting the `LogEvents` passed to this Appender

`destination` The output destination of the `Appender` in human-readable form (mainly for print output)

`filters`, `set_filters(filters)` a `list` that may contain `functions` or any R object with a `filter()` method. These functions must have exactly one argument: `event` which will get passed the LogEvent when the Filterable's `filter()` method is invoked. If all of these functions evaluate to `TRUE` the LogEvent is passed on. Since LogEvents have reference semantics, filters can also be abused to modify them before they are passed on. Look at the source code of `with_log_level()` or `with_log_value()` for examples.

Choosing the Right DBI Layout

Layouts for relational database tables are tricky as they have very strict column types and further restrictions. On top of that implementation details vary between database backends.

To make setting up `AppenderDbi` as painless as possible, the helper function `select_dbi_layout()` tries to automatically determine sensible LayoutDbi settings based on `conn` and - if it exists in the database already - `table`. If `table` does not exist in the database and you start logging, a new table will be created with the `col_types` from `layout`.

Methods

`flush()` Manually trigger flushing of the buffer

`show(n, threshold)` Show the last `n` log entries with a log level bellow `threshold`. The log entries will be formatted for console output via this Appenders Layout

`append(event)` Tell the Appender to process a LogEvent `event`. This method is usually not called by the user, but invoked by a Logger

`filter(event)` Determine whether the LogEvent `x` should be passed on to Appenders (`TRUE`) or not (`FALSE`). See also the active binding `filters`

`add_filter(filter, name = NULL)`, `remove_filter(pos)` Add or remove a filter. When adding a filter an optional `name` can be specified. `remove_filter()` can remove by position or name (if one was specified)

See Also

Other Appenders: `AppenderBuffer`, `AppenderConsole`, `AppenderFileRotating`, `AppenderFile`, `AppenderGmail`, `AppenderJson`, `AppenderPushbullet`, `AppenderRjdbc`, `AppenderSendmail`, `AppenderSyslog`, `AppenderTable`, `Appender`

AppenderDigest *Abstract class for digests*

Description

Abstract classes are exported for package developers that want to extend them, they cannot be instantiated directly.

Abstract class for Appenders that transmit digests of several log events at once, for example AppenderPushbullet, AppenderGmail and AppenderSendmail.

Fields

`subject_layout`, `set_layout(subject_layout)` Like `layout`, but used to format the subject/title of the digest. While `layout` is applied to each LogEvent of the digest, `subject_layout` is only applied to the last one.

`buffer_size`, `set_buffer_size(x)` `integer` scalar `>= 0` Number of LogEvents to buffer.

`buffer_events`, `buffer_df`, `buffer_dt` The contents of the buffer as a `list` of LogEvents, a `data.frame` or a `data.table`.

`flush_threshold`, `set_flush_threshold()` `integer` or `character` log level. Minimum event level that will trigger flushing of the buffer. This behaviour is implemented through `should_flush()`, and you can modify that function for different behaviour.

`should_flush(event)`, `set_should_flush(x)` A function with exactly one arguments: `event`. If the function returns `TRUE`, flushing of the buffer is triggered. Defaults to flushing if an event of level `error` or higher is registered.

`dt` Get the log recorded by this `Appender` as a `data.table` with a maximum of `buffer_size` rows

`data` Get the log recorded by this `Appender` as a `data.frame`

`threshold`, `set_threshold(level)` `character` or `integer` scalar. The minimum log level that triggers this logger. See log_levels

`layout`, `set_layout(layout)` a `Layout` that will be used for formatting the `LogEvents` passed to this Appender

`destination` The output destination of the `Appender` in human-readable form (mainly for print output)

`filters`, `set_filters(filters)` a `list` that may contain `functions` or any R object with a `filter()` method. These functions must have exactly one argument: `event` which will get passed the LogEvent when the Filterable's `filter()` method is invoked. If all of these functions evaluate to `TRUE` the LogEvent is passed on. Since LogEvents have reference semantics, filters can also be abused to modify them before they are passed on. Look at the source code of `with_log_level()` or `with_log_value()` for examples.

Methods

`flush()` Manually trigger flushing of the buffer

`show(n, threshold)` Show the last `n` log entries with a log level bellow `threshold`. The log entries will be formatted for console output via this Appenders Layout

`append(event)` Tell the Appender to process a LogEvent `event`. This method is usually not called by the user, but invoked by a Logger

`filter(event)` Determine whether the LogEvent `x` should be passed on to Appenders (`TRUE`) or not (`FALSE`). See also the active binding `filters`

`add_filter(filter, name = NULL)`, `remove_filter(pos)` Add or remove a filter. When adding a filter an optional `name` can be specified. `remove_filter()` can remove by position or name (if one was specified)

See Also

LayoutFormat, LayoutGlue

Other abstract classes: `AppenderMail`, `AppenderMemory`, `AppenderTable`, `Filterable`

AppenderDt *Log to an in-memory data.table*

Description

An Appender that outputs to an in-memory `data.table`. This kind of Appender is useful for interactive use, and has very little overhead.

Custom Fields

`AppenderDt` supports custom fields, but they have to be pre-allocated in the `prototype` argument. Custom fields that are not part of the prototype are discarded. If you want an Appender that retains all custom fields (at the cost of slightly less performance), take a look at AppenderBuffer.

With the default settings, the custom field `value` is included in the `data.table` as a list column to store arbitrary R objects (see example). It is recommended to use this feature only `TRACE` level.

Usage

```
x <- AppenderDt$new(threshold = NA_integer_, layout = LayoutFormat$new(fmt =
  "%L [%t] %m %f", timestamp_fmt = "%H:%M:%OS3", colors =
  getOption("lgr.colors", list())), prototype = data.table::data.table(.id =
  NA_integer_, level = NA_integer_, timestamp = Sys.time(), logger =
  NA_character_, caller = NA_character_, msg = NA_character_, .custom =
  list(list())), buffer_size = 1e+05, filters = NULL)

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

```
x$append(event)
x$filter(event)
x$format(color = FALSE, ...)
x$remove_filter(pos)
x$set_filters(filters)
x$set_layout(layout)
x$set_threshold(level)
x$show(threshold = NA_integer_, n = 20L)

x$data
x$destination
x$dt
x$filters
x$layout
x$threshold
```

Creating a Data Table Appender

In addition to the usual fields, `AppenderDt$new()` requires that you supply a `buffer_size` and a `prototype`. These determine the structure of the `data.table` used to store the log this appender creates and cannot be modified anymore after the instantiation of the appender.

The Layout for this Appender is used only to format console output of its `$show()` method.

buffer_size `integer` scalar. Number of rows of the in-memory `data.table`

prototype A prototype `data.table`. The prototype must be a `data.table` with the same columns and column types as the data you want to log. The actual content of the columns is irrelevant. There are a few columns that have special meaning, based on their name:

- `.id`: `integer` (mandatory). Must always be the first column and is used internally by the Appender
- `.custom`: `list` (optional). If present all custom values of the event (that are not already part of the prototype) are stored in this list column.

Fields

`dt` Get the log recorded by this `Appender` as a `data.table` with a maximum of `buffer_size` rows

`data` Get the log recorded by this `Appender` as a `data.frame`

`threshold`, `set_threshold(level)` `character` or `integer` scalar. The minimum log level that triggers this logger. See log_levels

`layout`, `set_layout(layout)` a `Layout` that will be used for formatting the `LogEvents` passed to this Appender

`destination` The output destination of the `Appender` in human-readable form (mainly for print output)

`filters`, `set_filters(filters)` a `list` that may contain `functions` or any R object with a `filter()` method. These functions must have exactly one argument: `event` which will get

passed the LogEvent when the Filterable's `filter()` method is invoked. If all of these functions evaluate to `TRUE` the LogEvent is passed on. Since LogEvents have reference semantics, filters can also be abused to modify them before they are passed on. Look at the source code of `with_log_level()` or `with_log_value()` for examples.

Methods

`show(n, threshold)` Show the last `n` log entries with a log level bellow `threshold`. The log entries will be formatted for console output via this Appenders Layout

`append(event)` Tell the Appender to process a LogEvent `event`. This method is usually not called by the user, but invoked by a Logger

`filter(event)` Determine whether the LogEvent `x` should be passed on to Appenders (`TRUE`) or not (`FALSE`). See also the active binding `filters`

`add_filter(filter, name = NULL)`, `remove_filter(pos)` Add or remove a filter. When adding a filter an optional `name` can be specified. `remove_filter()` can remove by position or name (if one was specified)

Comparison AppenderBuffer and AppenderDt

Both AppenderBuffer and AppenderDt do in memory buffering of events. AppenderBuffer retains a copies of the events it processes and has the ability to pass the buffered events on to other Appenders. AppenderDt converts the events to rows in a `data.table` and is a bit harder to configure. Used inside loops (several hundred iterations), AppenderDt has much less overhead than AppenderBuffer. For single logging calls and small loops, AppenderBuffer is more performant. This is related to how memory pre-allocation is handled by the appenders.

In short: Use AppenderDt if you want an in-memory log for interactive use, and AppenderBuffer if you actually want to buffer events

See Also

LayoutFormat, simple_logging, data.table::data.table

Examples

```
lg <- get_logger("test")
lg$config(list(
  appenders = list(memory = AppenderBuffer$new()),
  threshold = NA,
  propagate = FALSE  # to prevent routing to root logger for this example
))
lg$debug("test")
lg$error("test")

# Displaying the log
lg$appenders$memory$data
lg$appenders$memory$show()
show_log(target = lg$appenders$memory)
```

```
# If you pass a Logger to show_log(), it looks for the first AppenderDt
# that it can find.
show_log(target = lg)

# Custom fields are stored in the list column .custom by default
lg$info("the iris data frame", caps = LETTERS[1:5])
lg$appenders$memory$data
lg$appenders$memory$data$.custom[[3]]$caps
lg$config(NULL)
```

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 more or less a shortcut for `AppenderFile` with `LayoutJson` and a few extra methods for convenience.

Usage

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

x$add_filter(filter, name = NULL)
x$append(event)
x$filter(event)
x$format(color = FALSE, ...)
x$remove_filter(pos)
x$set_file(file)
x$set_filters(filters)
x$set_layout(layout)
x$set_threshold(level)
x$show(threshold = NA_integer_, n = 20L)

x$destination
x$file
x$filters
x$layout
x$threshold
```

Fields

`file`, `set_file(file)` `character` scalar. Path to the desired log file. If the file does not exist it will be created.

`threshold`, `set_threshold(level)` `character` or `integer` scalar. The minimum log level that triggers this logger. See log_levels

`layout`, `set_layout(layout)` a `Layout` that will be used for formatting the `LogEvents` passed to this Appender

`destination` The output destination of the `Appender` in human-readable form (mainly for print output)

`filters`, `set_filters(filters)` a `list` that may contain `functions` or any R object with a `filter()` method. These functions must have exactly one argument: `event` which will get passed the LogEvent when the Filterable's `filter()` method is invoked. If all of these functions evaluate to `TRUE` the LogEvent is passed on. Since LogEvents have reference semantics, filters can also be abused to modify them before they are passed on. Look at the source code of `with_log_level()` or `with_log_value()` for examples.

Creating a New Appender

New Appenders are instantiated with `<AppenderSubclass>$new()`. For the arguments to `new()` please refer to the section *Fields*. You can also modify those fields after the Appender has been created with setters in the form of `appender$set_<fieldname>(value)`

Methods

`append(event)` Tell the Appender to process a LogEvent `event`. This method is usually not called by the user, but invoked by a Logger

`filter(event)` Determine whether the LogEvent `x` should be passed on to Appenders (`TRUE`) or not (`FALSE`). See also the active binding `filters`

`add_filter(filter, name = NULL)`, `remove_filter(pos)` Add or remove a filter. When adding a filter an optional `name` can be specified. `remove_filter()` can remove by position or name (if one was specified)

See Also

LayoutFormat, LayoutJson

Other Appenders: `AppenderBuffer`, `AppenderConsole`, `AppenderDbi`, `AppenderFileRotating`, `AppenderGmail`, `AppenderJson`, `AppenderPushbullet`, `AppenderRjdbc`, `AppenderSendmail`, `AppenderSyslog`, `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)
```

AppenderFileRotating *Log to a rotating file*

Description

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

Usage

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

x <- 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, create_file = TRUE, cache_backups = TRUE)

x <- 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, create_file = TRUE, cache_backups = TRUE)

x$add_filter(filter, name = NULL)
x$append(event)
x$clone(deep = FALSE)
x$filter(event)
x$format(color = false, ...)
x$format(color = FALSE, ...)
x$prune(max_backups = self$max_backups)
x$remove_filter(pos)
x$rotate(force = FALSE, now = Sys.time())
x$rotate(force = FALSE)
x$set_age(x)
```

```
x$set_backup_dir(x)
x$set_cache_backups(x)
x$set_compression(x)
x$set_create_file(x)
x$set_file(file)
x$set_filters(filters)
x$set_fmt(x)
x$set_layout(layout)
x$set_max_backups(x)
x$set_overwrite(x)
x$set_size(x)
x$set_threshold(level)
x$show(threshold = NA_integer_, n = 20L)

x$age
x$backup_dir
x$backups
x$cache_backups
x$compression
x$create_file
x$destination
x$file
x$filters
x$fmt
x$layout
x$max_backups
x$overwrite
x$size
x$threshold
```

Fields

`age`, `size`, `max_backups`, `fmt`, `overwrite`, `compression`, `backup_dir` Please see `rotor::rotate()` for the meaning of these arguments (`fmt` is passed on as `format`).

`cache_backups`, `set_cache_backups(x)` `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.

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

`file`, `set_file(file)` `character` scalar. Path to the desired log file. If the file does not exist it will be created.

`threshold`, `set_threshold(level)` `character` or `integer` scalar. The minimum log level that triggers this logger. See log_levels

`layout`, `set_layout(layout)` a `Layout` that will be used for formatting the `LogEvents` passed to this Appender

`destination` The output destination of the `Appender` in human-readable form (mainly for print output)

`filters`, `set_filters(filters)` a `list` that may contain `functions` or any R object with a `filter()` method. These functions must have exactly one argument: `event` which will get passed the LogEvent when the Filterable's `filter()` method is invoked. If all of these functions evaluate to `TRUE` the LogEvent is passed on. Since LogEvents have reference semantics, filters can also be abused to modify them before they are passed on. Look at the source code of `with_log_level()` or `with_log_value()` for examples.

Creating a New Appender

New Appenders are instantiated with `<AppenderSubclass>$new()`. For the arguments to `new()` please refer to the section *Fields*. You can also modify those fields after the Appender has been created with setters in the form of `appender$set_<fieldname>(value)`

Methods

`append(event)` Tell the Appender to process a LogEvent `event`. This method is usually not called by the user, but invoked by a Logger

`filter(event)` Determine whether the LogEvent `x` should be passed on to Appenders (`TRUE`) or not (`FALSE`). See also the active binding `filters`

`add_filter(filter, name = NULL)`, `remove_filter(pos)` Add or remove a filter. When adding a filter an optional `name` can be specified. `remove_filter()` can remove by position or name (if one was specified)

See Also

LayoutFormat, LayoutJson, `rotor::rotate()`

Other Appenders: `AppenderBuffer`, `AppenderConsole`, `AppenderDbi`, `AppenderFile`, `AppenderGmail`, `AppenderJson`, `AppenderPushbullet`, `AppenderRjdbc`, `AppenderSendmail`, `AppenderSyslog`, `AppenderTable`, `Appender`

AppenderGmail *Send emails via gmailr*

Description

Send mails via `gmailr::send_message()`. This Appender keeps an in-memory buffer like AppenderBuffer. If the buffer is flushed, usually because an event of specified magnitude is encountered, all buffered events are concatenated to a single message. The default behaviour is to push the last 30 log events in case a `fatal` event is encountered.

Usage

```
x <- AppenderGmail$new(to, threshold = NA_integer_, flush_threshold = "fatal",
  layout = LayoutFormat$new(fmt = "%L [%t] %m %f", timestamp_fmt = "%H:%M:%S"),
  subject_layout = LayoutFormat$new(fmt = "[LGR] %L: %m"), buffer_size = 30,
  from = get_user(), cc = NULL, bcc = NULL, html = FALSE, filters = NULL)

x$add_filter(filter, name = NULL)
x$append(event)
x$filter(event)
x$flush()
x$format(color = FALSE, ...)
x$remove_filter(pos)
x$set_bcc(x)
x$set_buffer_size(x)
x$set_cc(x)
x$set_filters(filters)
x$set_flush_threshold(level)
x$set_from(x)
x$set_html(x)
x$set_layout(layout)
x$set_should_flush(x)
x$set_subject_layout(layout)
x$set_threshold(level)
x$set_to(x)
x$show(threshold = NA_integer_, n = 20L)

x$bcc
x$buffer_df
x$buffer_dt
x$buffer_events
x$buffer_size
x$cc
x$data
x$destination
x$dt
x$filters
x$flush_threshold
x$from
x$html
x$layout
x$should_flush
x$subject_layout
x$threshold
x$to
```

Fields

`to`, `from`, `cc`, `bcc` `character` vectors.

`html`, `set_html()` `TRUE` or `FALSE`. Send a html email message? This does currently only formats the log contents as monospace verbatim text.

`subject_layout`, `set_layout(subject_layout)` Like `layout`, but used to format the subject/title of the digest. While `layout` is applied to each LogEvent of the digest, `subject_layout` is only applied to the last one.

`buffer_size`, `set_buffer_size(x)` `integer` scalar `>= 0` Number of LogEvents to buffer.

`buffer_events`, `buffer_df`, `buffer_dt` The contents of the buffer as a `list` of LogEvents, a `data.frame` or a `data.table`.

`flush_threshold`, `set_flush_threshold()` `integer` or `character` log level. Minimum event level that will trigger flushing of the buffer. This behaviour is implemented through `should_flush()`, and you can modify that function for different behaviour.

`should_flush(event)`, `set_should_flush(x)` A function with exactly one arguments: `event`. If the function returns `TRUE`, flushing of the buffer is triggered. Defaults to flushing if an event of level `error` or higher is registered.

`dt` Get the log recorded by this `Appender` as a `data.table` with a maximum of `buffer_size` rows

`data` Get the log recorded by this `Appender` as a `data.frame`

`threshold`, `set_threshold(level)` `character` or `integer` scalar. The minimum log level that triggers this logger. See log_levels

`layout`, `set_layout(layout)` a `Layout` that will be used for formatting the `LogEvents` passed to this Appender

`destination` The output destination of the `Appender` in human-readable form (mainly for print output)

`filters`, `set_filters(filters)` a `list` that may contain `functions` or any R object with a `filter()` method. These functions must have exactly one argument: `event` which will get passed the LogEvent when the Filterable's `filter()` method is invoked. If all of these functions evaluate to `TRUE` the LogEvent is passed on. Since LogEvents have reference semantics, filters can also be abused to modify them before they are passed on. Look at the source code of `with_log_level()` or `with_log_value()` for examples.

Methods

`flush()` Manually trigger flushing of the buffer

`show(n, threshold)` Show the last `n` log entries with a log level bellow `threshold`. The log entries will be formatted for console output via this Appenders Layout

`append(event)` Tell the Appender to process a LogEvent `event`. This method is usually not called by the user, but invoked by a Logger

`filter(event)` Determine whether the LogEvent `x` should be passed on to Appenders (`TRUE`) or not (`FALSE`). See also the active binding `filters`

`add_filter(filter, name = NULL)`, `remove_filter(pos)` Add or remove a filter. When adding a filter an optional `name` can be specified. `remove_filter()` can remove by position or name (if one was specified)

See Also

LayoutFormat, LayoutGlue

Other Appenders: `AppenderBuffer`, `AppenderConsole`, `AppenderDbi`, `AppenderFileRotating`, `AppenderFile`, `AppenderJson`, `AppenderPushbullet`, `AppenderRjdbc`, `AppenderSendmail`, `AppenderSyslog`, `AppenderTable`, `Appender`

`AppenderJson` *Log to a JSON file*

Description

`AppenderJson` is a shortcut for `AppenderFile` with `LayoutJson`, but comes with an extra method `show()` and an extra active field `data` to comfortably access the underlying file.

Usage

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

x$add_filter(filter, name = NULL)
x$append(event)
x$filter(event)
x$format(color = FALSE, ...)
x$remove_filter(pos)
x$set_file(file)
x$set_filters(filters)
x$set_layout(layout)
x$set_threshold(level)
x$show(threshold = NA_integer_, n = 20L)

x$destination
x$file
x$filters
x$layout
x$threshold
```

Creating a New Appender

New Appenders are instantiated with `<AppenderSubclass>$new()`. For the arguments to `new()` please refer to the section *Fields*. You can also modify those fields after the Appender has been created with setters in the form of `appender$set_<fieldname>(value)`

Fields

`file`, `set_file(file)` `character` scalar. Path to the desired log file. If the file does not exist it will be created.

`threshold`, `set_threshold(level)` `character` or `integer` scalar. The minimum log level that triggers this logger. See log_levels

`layout`, `set_layout(layout)` a `Layout` that will be used for formatting the `LogEvents` passed to this Appender

`destination` The output destination of the `Appender` in human-readable form (mainly for print output)

`filters`, `set_filters(filters)` a `list` that may contain `functions` or any R object with a `filter()` method. These functions must have exactly one argument: `event` which will get passed the LogEvent when the Filterable's `filter()` method is invoked. If all of these functions evaluate to `TRUE` the LogEvent is passed on. Since LogEvents have reference semantics, filters can also be abused to modify them before they are passed on. Look at the source code of `with_log_level()` or `with_log_value()` for examples.

`data` Get the log recorded by this `Appender` as a `data.frame`

`threshold`, `set_threshold(level)` `character` or `integer` scalar. The minimum log level that triggers this logger. See log_levels

`layout`, `set_layout(layout)` a `Layout` that will be used for formatting the `LogEvents` passed to this Appender

`destination` The output destination of the `Appender` in human-readable form (mainly for print output)

`filters`, `set_filters(filters)` a `list` that may contain `functions` or any R object with a `filter()` method. These functions must have exactly one argument: `event` which will get passed the LogEvent when the Filterable's `filter()` method is invoked. If all of these functions evaluate to `TRUE` the LogEvent is passed on. Since LogEvents have reference semantics, filters can also be abused to modify them before they are passed on. Look at the source code of `with_log_level()` or `with_log_value()` for examples.

Methods

`show(n, threshold)` Show the last `n` log entries with a log level bellow `threshold`. The log entries will be formatted as in the source JSON file

`append(event)` Tell the Appender to process a LogEvent `event`. This method is usually not called by the user, but invoked by a Logger

`filter(event)` Determine whether the LogEvent `x` should be passed on to Appenders (`TRUE`) or not (`FALSE`). See also the active binding `filters`

`add_filter(filter, name = NULL)`, `remove_filter(pos)` Add or remove a filter. When adding a filter an optional `name` can be specified. `remove_filter()` can remove by position or name (if one was specified)

See Also

LayoutFormat, LayoutJson

Other Appenders: AppenderBuffer, AppenderConsole, AppenderDbi, AppenderFileRotating, AppenderFile, AppenderGmail, AppenderPushbullet, AppenderRjdbc, AppenderSendmail, AppenderSyslog, AppenderTable, Appender

Examples

```
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)
```

AppenderMemory *Abstract class for logging to memory buffers*

Description

Abstract classes are exported for package developers that want to extend them, they cannot be instantiated directly.

AppenderMemory is extended by Appenders that retain an in-memory event buffer, such as AppenderBuffer and AppenderPushbullet.

Usage

```
x <- AppenderMemory$new(layout = Layout$new(), threshold = NA_integer_)

x$add_filter(filter, name = NULL)
x$append(event)
x$filter(event)
x$flush()
x$format(color = FALSE, ...)
x$remove_filter(pos)
x$set_buffer_size(x)
x$set_filters(filters)
x$set_flush_on_exit(x)
x$set_flush_on_rotate(x)
```

```
x$set_flush_threshold(level)
x$set_layout(layout)
x$set_should_flush(x)
x$set_threshold(level)
x$show(threshold = NA_integer_, n = 20L)

x$buffer_df
x$buffer_dt
x$buffer_events
x$buffer_size
x$data
x$destination
x$dt
x$filters
x$flush_on_exit
x$flush_on_rotate
x$flush_threshold
x$layout
x$should_flush
x$threshold
```

Fields

`buffer_size`, `set_buffer_size(x)` `integer` scalar `>= 0` Number of LogEvents to buffer.

`buffer_events`, `buffer_df`, `buffer_dt` The contents of the buffer as a `list` of LogEvents, a `data.frame` or a `data.table`.

`flush_threshold`, `set_flush_threshold()` `integer` or `character` log level. Minimum event level that will trigger flushing of the buffer. This behaviour is implemented through `should_flush()`, and you can modify that function for different behaviour.

`should_flush(event)`, `set_should_flush(x)` A function with exactly one arguments: `event`. If the function returns `TRUE`, flushing of the buffer is triggered. Defaults to flushing if an event of level `error` or higher is registered.

`dt` Get the log recorded by this `Appender` as a `data.table` with a maximum of `buffer_size` rows

`data` Get the log recorded by this `Appender` as a `data.frame`

`threshold`, `set_threshold(level)` `character` or `integer` scalar. The minimum log level that triggers this logger. See log_levels

`layout`, `set_layout(layout)` a `Layout` that will be used for formatting the `LogEvents` passed to this Appender

`destination` The output destination of the `Appender` in human-readable form (mainly for print output)

`filters`, `set_filters(filters)` a `list` that may contain `functions` or any R object with a `filter()` method. These functions must have exactly one argument: `event` which will get

passed the LogEvent when the Filterable's `filter()` method is invoked. If all of these functions evaluate to `TRUE` the LogEvent is passed on. Since LogEvents have reference semantics, filters can also be abused to modify them before they are passed on. Look at the source code of `with_log_level()` or `with_log_value()` for examples.

Methods

`flush()` Manually trigger flushing of the buffer

`show(n, threshold)` Show the last `n` log entries with a log level bellow `threshold`. The log entries will be formatted for console output via this Appenders Layout

`append(event)` Tell the Appender to process a LogEvent `event`. This method is usually not called by the user, but invoked by a Logger

`filter(event)` Determine whether the LogEvent `x` should be passed on to Appenders (`TRUE`) or not (`FALSE`). See also the active binding `filters`

`add_filter(filter, name = NULL)`, `remove_filter(pos)` Add or remove a filter. When adding a filter an optional `name` can be specified. `remove_filter()` can remove by position or name (if one was specified)

See Also

LayoutFormat

Other abstract classes: `AppenderDigest`, `AppenderMail`, `AppenderTable`, `Filterable`

`AppenderPushbullet` *Send push-notifications via RPushbullet*

Description

Send push notifications via pushbullet. This Appender keeps an in-memory buffer like AppenderBuffer. If the buffer is flushed, usually because an event of specified magnitude is encountered, all buffered events are concatenated to a single message that is sent to `RPushbullet::pbPost()`. The default behaviour is to push the last 7 log events in case a `fatal` event is encountered.

Usage

```
x <- AppenderPushbullet$new(threshold = NA_integer_, flush_threshold = "fatal",
  layout = LayoutFormat$new(fmt = "%K %t> %m %f", timestamp_fmt = "%H:%M:%S"),
  subject_layout = LayoutFormat$new(fmt = "[LGR] %L: %m"), buffer_size = 6,
  recipients = NULL, email = NULL, channel = NULL, devices = NULL, apikey =
  NULL, filters = NULL)

x$add_filter(filter, name = NULL)
x$append(event)
x$filter(event)
```

```
x$flush()
x$format(color = FALSE, ...)
x$remove_filter(pos)
x$set_apikey(x)
x$set_buffer_size(x)
x$set_channel(x)
x$set_devices(x)
x$set_email(x)
x$set_filters(filters)
x$set_flush_threshold(level)
x$set_layout(layout)
x$set_recipients(x)
x$set_should_flush(x)
x$set_subject_layout(layout)
x$set_threshold(level)
x$show(threshold = NA_integer_, n = 20L)

x$apikey
x$buffer_df
x$buffer_dt
x$buffer_events
x$buffer_size
x$channel
x$data
x$destination
x$devices
x$dt
x$email
x$filters
x$flush_threshold
x$layout
x$recipients
x$should_flush
x$subject_layout
x$threshold
```

Fields

`apikey`, `recipients`, `email`, `channel`, `devices` See `RPushbullet::pbPost()`

`buffer_size`, `set_buffer_size(x)` `integer` scalar `>= 0` Number of LogEvents to buffer.

`buffer_events`, `buffer_df`, `buffer_dt` The contents of the buffer as a `list` of LogEvents, a `data.frame` or a `data.table`.

`flush_threshold`, `set_flush_threshold()` `integer` or `character` log level. Minimum event level that will trigger flushing of the buffer. This behaviour is implemented through `should_flush()`, and you can modify that function for different behaviour.

`should_flush(event)`, `set_should_flush(x)` A function with exactly one arguments: `event`. If the function returns `TRUE`, flushing of the buffer is triggered. Defaults to flushing if an event of level `error` or higher is registered.

`dt` Get the log recorded by this `Appender` as a `data.table` with a maximum of `buffer_size` rows

`data` Get the log recorded by this `Appender` as a `data.frame`

`threshold`, `set_threshold(level)` `character` or `integer` scalar. The minimum log level that triggers this logger. See log_levels

`layout`, `set_layout(layout)` a `Layout` that will be used for formatting the `LogEvents` passed to this Appender

`destination` The output destination of the `Appender` in human-readable form (mainly for print output)

`filters`, `set_filters(filters)` a `list` that may contain `functions` or any R object with a `filter()` method. These functions must have exactly one argument: `event` which will get passed the LogEvent when the Filterable's `filter()` method is invoked. If all of these functions evaluate to `TRUE` the LogEvent is passed on. Since LogEvents have reference semantics, filters can also be abused to modify them before they are passed on. Look at the source code of `with_log_level()` or `with_log_value()` for examples.

Methods

`flush()` Manually trigger flushing of the buffer

`show(n, threshold)` Show the last `n` log entries with a log level bellow `threshold`. The log entries will be formatted for console output via this Appenders Layout

`append(event)` Tell the Appender to process a LogEvent `event`. This method is usually not called by the user, but invoked by a Logger

`filter(event)` Determine whether the LogEvent `x` should be passed on to Appenders (`TRUE`) or not (`FALSE`). See also the active binding `filters`

`add_filter(filter, name = NULL)`, `remove_filter(pos)` Add or remove a filter. When adding a filter an optional `name` can be specified. `remove_filter()` can remove by position or name (if one was specified)

See Also

LayoutFormat, LayoutGlue

Other Appenders: `AppenderBuffer`, `AppenderConsole`, `AppenderDbi`, `AppenderFileRotating`, `AppenderFile`, `AppenderGmail`, `AppenderJson`, `AppenderRjdbc`, `AppenderSendmail`, `AppenderSyslog`, `AppenderTable`, `Appender`

AppenderRjdbc *Log to databases via RJDBC*

Description

Log to a database table with the **RJDBC** package. **RJDBC** is only somewhat **DBI** compliant and does not work with AppenderDbi. **I do not recommend using RJDBC if it can be avoided.**. AppenderRjdbc is only tested for DB2 databases, and it is likely it will not work properly for other databases. Please file a bug report if you encounter any issues.

Usage

```
x <- AppenderRjdbc$new(conn, table, threshold = NA_integer_, layout =
  select_dbi_layout(conn, table), close_on_exit = TRUE, buffer_size = 10,
  flush_threshold = "error", flush_on_exit = TRUE, flush_on_rotate = TRUE,
  should_flush = default_should_flush, filters = NULL)

x$add_filter(filter, name = NULL)
x$append(event)
x$filter(event)
x$flush()
x$format(color = FALSE, ...)
x$remove_filter(pos)
x$set_buffer_size(x)
x$set_close_on_exit(x)
x$set_conn(conn)
x$set_filters(filters)
x$set_flush_on_exit(x)
x$set_flush_on_rotate(x)
x$set_flush_threshold(level)
x$set_layout(layout)
x$set_should_flush(x)
x$set_threshold(level)
x$show(threshold = NA_integer_, n = 20)
x$show(threshold = NA_integer_, n = 20L)

x$buffer_df
x$buffer_dt
x$buffer_events
x$buffer_size
x$close_on_exit
x$col_types
x$conn
x$data
x$destination
x$dt
x$filters
```

```
x$flush_on_exit
x$flush_on_rotate
x$flush_threshold
x$layout
x$should_flush
x$table
x$table_id
x$table_name
x$threshold
```

Fields

Note: `$data` and `show()` query the data from the remote database and might be slow for very large logs.

`close_on_exit`, `set_close_on_exit()` `TRUE` or `FALSE`. Close the Database connection when the Logger is removed?

`conn`, `set_conn(conn)` a DBI connection

`table` Name of the target database table

`buffer_size`, `set_buffer_size(x)` `integer` scalar `>= 0` Number of LogEvents to buffer.

`buffer_events`, `buffer_df`, `buffer_dt` The contents of the buffer as a `list` of LogEvents, a `data.frame` or a `data.table`.

`flush_threshold`, `set_flush_threshold()` `integer` or `character` log level. Minimum event level that will trigger flushing of the buffer. This behaviour is implemented through `should_flush()`, and you can modify that function for different behaviour.

`should_flush(event)`, `set_should_flush(x)` A function with exactly one arguments: `event`. If the function returns `TRUE`, flushing of the buffer is triggered. Defaults to flushing if an event of level `error` or higher is registered.

`dt` Get the log recorded by this `Appender` as a `data.table` with a maximum of `buffer_size` rows

`data` Get the log recorded by this `Appender` as a `data.frame`

`threshold`, `set_threshold(level)` `character` or `integer` scalar. The minimum log level that triggers this logger. See log_levels

`layout`, `set_layout(layout)` a `Layout` that will be used for formatting the `LogEvents` passed to this Appender

`destination` The output destination of the `Appender` in human-readable form (mainly for print output)

`filters`, `set_filters(filters)` a `list` that may contain `functions` or any R object with a `filter()` method. These functions must have exactly one argument: `event` which will get passed the LogEvent when the Filterable's `filter()` method is invoked. If all of these functions evaluate to `TRUE` the LogEvent is passed on. Since LogEvents have reference semantics, filters can also be abused to modify them before they are passed on. Look at the source code of `with_log_level()` or `with_log_value()` for examples.

Methods

`flush()` Manually trigger flushing of the buffer

`show(n, threshold)` Show the last `n` log entries with a log level bellow `threshold`. The log entries will be formatted for console output via this Appenders Layout

`append(event)` Tell the Appender to process a LogEvent `event`. This method is usually not called by the user, but invoked by a Logger

`filter(event)` Determine whether the LogEvent `x` should be passed on to Appenders (`TRUE`) or not (`FALSE`). See also the active binding `filters`

`add_filter(filter, name = NULL)`, `remove_filter(pos)` Add or remove a filter. When adding a filter an optional `name` can be specified. `remove_filter()` can remove by position or name (if one was specified)

Creating a New Appender

An AppenderDbi is linked to a database table via its `table` argument. If the table does not exist it is created either when the Appender is first instantiated or (more likely) when the first LogEvent would be written to that table. Rather than to rely on this feature, it is recommended that you create the target log table first manually using an `SQL CREATE TABLE` statement as this is safer and more flexible. See also LayoutDbi.

New Appenders are instantiated with `<AppenderSubclass>$new()`. For the arguments to `new()` please refer to the section *Fields*. You can also modify those fields after the Appender has been created with setters in the form of `appender$set_<fieldname>(value)`

Choosing the Right DBI Layout

Layouts for relational database tables are tricky as they have very strict column types and further restrictions. On top of that implementation details vary between database backends.

To make setting up `AppenderDbi` as painless as possible, the helper function `select_dbi_layout()` tries to automatically determine sensible LayoutDbi settings based on `conn` and - if it exists in the database already - `table`. If `table` does not exist in the database and you start logging, a new table will be created with the `col_types` from `layout`.

See Also

LayoutFormat, simple_logging, data.table::data.table

Other Appenders: `AppenderBuffer`, `AppenderConsole`, `AppenderDbi`, `AppenderFileRotating`, `AppenderFile`, `AppenderGmail`, `AppenderJson`, `AppenderPushbullet`, `AppenderSendmail`, `AppenderSyslog`, `AppenderTable`, `Appender`

AppenderSendmail *Send emails via sendmailR*

Description

Send mails via `sendmailR::sendmail()`, which requires that you have access to an SMTP server that does not require authentication. This Appender keeps an in-memory buffer like AppenderBuffer. If the buffer is flushed, usually because an event of specified magnitude is encountered, all buffered events are concatenated to a single message. The default behaviour is to push the last 30 log events in case a `fatal` event is encountered.

Usage

```
x <- AppenderSendmail$new(to, control, threshold = NA_integer_, flush_threshold
  = "fatal", layout = LayoutFormat$new(fmt = " %L [%t] %m %f", timestamp_fmt =
  "%H:%M:%S"), subject_layout = LayoutFormat$new(fmt = "[LGR] %L: %m"),
  buffer_size = 29, from = get_user(), cc = NULL, bcc = NULL, html = FALSE,
  headers = NULL, filters = NULL)

x$add_filter(filter, name = NULL)
x$append(event)
x$filter(event)
x$flush()
x$format(color = FALSE, ...)
x$remove_filter(pos)
x$set_bcc(x)
x$set_buffer_size(x)
x$set_cc(x)
x$set_control(x)
x$set_filters(filters)
x$set_flush_threshold(level)
x$set_from(x)
x$set_headers(x)
x$set_html(x)
x$set_layout(layout)
x$set_should_flush(x)
x$set_subject_layout(layout)
x$set_threshold(level)
x$set_to(x)
x$show(threshold = NA_integer_, n = 20L)

x$bcc
x$buffer_df
x$buffer_dt
x$buffer_events
x$buffer_size
x$cc
```

```
x$control
x$data
x$destination
x$dt
x$filters
x$flush_threshold
x$from
x$headers
x$html
x$layout
x$should_flush
x$subject_layout
x$threshold
x$to
```

Fields

`headers`, `control` see `sendmailR::sendmail()`

`to`, `from`, `cc`, `bcc` `character` vectors.

`html`, `set_html()` `TRUE` or `FALSE`. Send a html email message? This does currently only formats the log contents as monospace verbatim text.

`subject_layout`, `set_layout(subject_layout)` Like `layout`, but used to format the subject/title of the digest. While `layout` is applied to each LogEvent of the digest, `subject_layout` is only applied to the last one.

`buffer_size`, `set_buffer_size(x)` `integer` scalar `>= 0` Number of LogEvents to buffer.

`buffer_events`, `buffer_df`, `buffer_dt` The contents of the buffer as a `list` of LogEvents, a `data.frame` or a `data.table`.

`flush_threshold`, `set_flush_threshold()` `integer` or `character` log level. Minimum event level that will trigger flushing of the buffer. This behaviour is implemented through `should_flush()`, and you can modify that function for different behaviour.

`should_flush(event)`, `set_should_flush(x)` A function with exactly one arguments: `event`. If the function returns `TRUE`, flushing of the buffer is triggered. Defaults to flushing if an event of level `error` or higher is registered.

`dt` Get the log recorded by this `Appender` as a `data.table` with a maximum of `buffer_size` rows

`data` Get the log recorded by this `Appender` as a `data.frame`

`threshold`, `set_threshold(level)` `character` or `integer` scalar. The minimum log level that triggers this logger. See log_levels

`layout`, `set_layout(layout)` a `Layout` that will be used for formatting the `LogEvents` passed to this Appender

`destination` The output destination of the `Appender` in human-readable form (mainly for print output)

`filters`, `set_filters(filters)` a `list` that may contain `functions` or any R object with a `filter()` method. These functions must have exactly one argument: `event` which will get passed the LogEvent when the Filterable's `filter()` method is invoked. If all of these functions evaluate to `TRUE` the LogEvent is passed on. Since LogEvents have reference semantics, filters can also be abused to modify them before they are passed on. Look at the source code of `with_log_level()` or `with_log_value()` for examples.

Methods

`flush()` Manually trigger flushing of the buffer

`show(n, threshold)` Show the last `n` log entries with a log level bellow `threshold`. The log entries will be formatted for console output via this Appenders Layout

`append(event)` Tell the Appender to process a LogEvent `event`. This method is usually not called by the user, but invoked by a Logger

`filter(event)` Determine whether the LogEvent `x` should be passed on to Appenders (`TRUE`) or not (`FALSE`). See also the active binding `filters`

`add_filter(filter, name = NULL)`, `remove_filter(pos)` Add or remove a filter. When adding a filter an optional `name` can be specified. `remove_filter()` can remove by position or name (if one was specified)

Note

The default Layout's `fmt` indents each log entry with 3 blanks. This is a workaround so that Microsoft Outlook does not mess up the line breaks.

See Also

LayoutFormat, LayoutGlue

Other Appenders: `AppenderBuffer`, `AppenderConsole`, `AppenderDbi`, `AppenderFileRotating`, `AppenderFile`, `AppenderGmail`, `AppenderJson`, `AppenderPushbullet`, `AppenderRjdbc`, `AppenderSyslog`, `AppenderTable`, `Appender`

AppenderSyslog *Log to the POSIX System Log*

Description

An Appender that writes to the syslog on supported POSIX platforms. Requires the **rsyslog** package.

Usage

```
x <- AppenderSyslog$new(identifier = NULL, threshold = NA_integer_, layout =
  LayoutFormat$new("%m"), filters = NULL, syslog_levels = c(CRITICAL = "fatal",
  ERR = "error", WARNING = "warn", INFO = "info", DEBUG = "debug", DEBUG =
  "trace"))

x$add_filter(filter, name = NULL)
x$append(event)
x$filter(event)
x$format(color = FALSE, ...)
x$remove_filter(pos)
x$set_filters(filters)
x$set_identifier(x)
x$set_layout(layout)
x$set_syslog_levels(x)
x$set_threshold(level)

x$destination
x$filters
x$identifier
x$layout
x$syslog_levels
x$threshold
```

Fields

`identifier` `character` scalar. A string identifying the process; if `NULL` defaults to the logger name

`syslog_levels` • a named `character` vector mapping whose names are log levels as understood by `rsyslog::syslog()` and whose values are lgr log levels (either `character` or `numeric`)

• a `function` that takes a vector of lgr log levels as input and returns a `character` vector of log levels for `rsyslog::syslog()`.

`threshold`, `set_threshold(level)` `character` or `integer` scalar. The minimum log level that triggers this logger. See log_levels

`layout`, `set_layout(layout)` a `Layout` that will be used for formatting the `LogEvents` passed to this Appender

`destination` The output destination of the `Appender` in human-readable form (mainly for print output)

`filters`, `set_filters(filters)` a `list` that may contain `functions` or any R object with a `filter()` method. These functions must have exactly one argument: `event` which will get passed the LogEvent when the Filterable's `filter()` method is invoked. If all of these functions evaluate to `TRUE` the LogEvent is passed on. Since LogEvents have reference semantics, filters can also be abused to modify them before they are passed on. Look at the source code of `with_log_level()` or `with_log_value()` for examples.

Creating a New Appender

New Appenders are instantiated with `<AppenderSubclass>$new()`. For the arguments to `new()` please refer to the section *Fields*. You can also modify those fields after the Appender has been created with setters in the form of `appender$set_<fieldname>(value)`

Methods

`append(event)` Tell the Appender to process a LogEvent `event`. This method is usually not called by the user, but invoked by a Logger

`filter(event)` Determine whether the LogEvent `x` should be passed on to Appenders (`TRUE`) or not (`FALSE`). See also the active binding `filters`

`add_filter(filter, name = NULL)`, `remove_filter(pos)` Add or remove a filter. When adding a filter an optional `name` can be specified. `remove_filter()` can remove by position or name (if one was specified)

See Also

LayoutFormat, LayoutJson

Other Appenders: `AppenderBuffer`, `AppenderConsole`, `AppenderDbi`, `AppenderFileRotating`, `AppenderFile`, `AppenderGmail`, `AppenderJson`, `AppenderPushbullet`, `AppenderRjdbc`, `AppenderSendmail`, `AppenderTable`, `Appender`

Examples

```
if (requireNamespace("rsyslog", quietly = TRUE)) {
  lg <- get_logger("rsyslog/test")
  lg$add_appender(AppenderSyslog$new(), "syslog")
  lg$info("A test message")

  if (Sys.info()[["sysname"]] == "Linux"){
    system("journalctl -t 'rsyslog/test'")
  }

  invisible(lg$config(NULL))  # cleanup
}
```

AppenderTable *Abstract class for logging to tabular structures*

Description

Abstract classes are exported for package developers that want to extend them, they cannot be instantiated directly.

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.

Fields

`data` Get the log recorded by this `Appender` as a `data.frame`

`threshold`, `set_threshold(level)` `character` or `integer` scalar. The minimum log level that triggers this logger. See log_levels

`layout`, `set_layout(layout)` a `Layout` that will be used for formatting the `LogEvents` passed to this Appender

`destination` The output destination of the `Appender` in human-readable form (mainly for print output)

`filters`, `set_filters(filters)` a `list` that may contain `functions` or any R object with a `filter()` method. These functions must have exactly one argument: `event` which will get passed the LogEvent when the Filterable's `filter()` method is invoked. If all of these functions evaluate to `TRUE` the LogEvent is passed on. Since LogEvents have reference semantics, filters can also be abused to modify them before they are passed on. Look at the source code of `with_log_level()` or `with_log_value()` for examples.

Methods

`show(n, threshold)` Show the last `n` log entries with a log level bellow `threshold`.

`append(event)` Tell the Appender to process a LogEvent `event`. This method is usually not called by the user, but invoked by a Logger

`filter(event)` Determine whether the LogEvent `x` should be passed on to Appenders (`TRUE`) or not (`FALSE`). See also the active binding `filters`

`add_filter(filter, name = NULL)`, `remove_filter(pos)` Add or remove a filter. When adding a filter an optional `name` can be specified. `remove_filter()` can remove by position or name (if one was specified)

See Also

Other abstract classes: `AppenderDigest`, `AppenderMail`, `AppenderMemory`, `Filterable`

Other Appenders: `AppenderBuffer`, `AppenderConsole`, `AppenderDbi`, `AppenderFileRotating`, `AppenderFile`, `AppenderGmail`, `AppenderJson`, `AppenderPushbullet`, `AppenderRjdbc`, `AppenderSendmail`, `AppenderSyslog`, `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, ...)

as.data.table.LogEvent(x, ...)

as_tibble.LogEvent(x, ...)
```

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 base::as.data.frame()) and is only included for compatibility.
...	passed on to data.frame()

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]])
```

---

basic_config *Basic Setup for the Logging System*

Description

A quick and easy way to configure the root logger. This is less powerful then using lgr$config() or lgr$set_*(), 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 AppenderFile will be created that logs to this file. If the filename ends in .jsonl, the Appender will be set up to use the JSON Lines format instead of plain text (see AppenderFile and AppenderJson).
fmt	character scalar: Format to use if file is supplied and not a .jsonl file. If NULL it defaults to "%L [%t] %m" (see format.LogEvent)
timestamp_fmt	see format.POSIXct()
threshold	character or integer scalar. The minimum log level that should be processed by the root logger.
appenders	a single Appender 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 fmt but used for console output
console_timestamp_fmt	character scalar: like timestamp_fmt but used for console output
memory	logical scalar. or a threshold (see above). Add an Appender that logs to a memory buffer, see also show_log() and AppenderBuffer

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()
```

colorize_levels *Colorize Levels*

Description

Colorize Levels

Usage

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

Arguments

x	numeric or character levels to be colored. Unlike in many other functions in lgr, character levels are *not* case sensitive in this function and leading/trailing whitespace is ignored to make it more comfortable to use colorize_levels() inside formatting functions.
colors	A list of functions that will be used to color the log levels (likely from crayon::crayon).

Value

a character vector wit color ANSI codes

See Also

Other formatting utils: label_levels

Examples

```
cat(colorize_levels(c(100, 200)))
cat(colorize_levels(c("trace", "warn ", "DEBUG")))
```

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)
```

---

default_should_flush *Default should_flush function*

Description

This is the default "should_flush()" trigger function for Appenders that support such a mechanism, such as AppenderBuffer and AppenderDbi. It returns TRUE if the event's level meets or exceeds the Appender's flush_threshold.

Usage

```
default_should_flush(event)
```

Arguments

event	a LogEvent

Value

TRUE or FALSE

EventFilter *Event Filters*

Description

Filters can be used for the $set_filter() and $add_filter() methods of Appenders and Loggers. You normally do not need to construct a formal EventFilter object, you can just use any function that has the single argument event or any object that has a filter method.

Modifying LogEvents with Filters

Since LogEvents are R6 objects with reference semantics, Filters can also be abused to modify log events before passing them on. lgr comes with a few preset filters that use this property:

`FilterInject$new(..., .list)` `...` and `.list` can take any number of named R6 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()`

`FilterForceLevel$new(level)` Sets the level of all LogEvents processed by the Appender/Logger that this filter is attached to to `level`. See also `with_log_value()`

Accessing Appenders and Loggers from Filters

You can use the special function `.obj()` to access the calling Logger/Appender from within a filter

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.

Examples

```
# using filters to modify log events
lg <- get_logger("test")

analyse <- function(){
  lg$add_filter(FilterForceLevel$new("info"), "force")
  lg$add_filter(FilterInject$new(type = "analysis"), "inject")
  on.exit(lg$remove_filter(c("force", "inject")))
  lg$debug("a debug message")
  lg$error("an error")
}
analyse()
lg$error("an error")
lg$config(NULL)  # reset config


# using .obj()
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)
```

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 R6ClassGenerator 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 $config(NULL) 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)
# this invalidates references to the Logger
try(lg$info("lg has been invalidated an no longer works"))

# we have to recreate it
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 *Check if an R Object is a Filter*

Description

Check if an R Object is a Filter

Usage

```
is_filter(x)
```

Arguments

x	any R Object

See Also

EventFilter

`label_levels` *Label/Unlabel Log Levels*

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`	a named `integer` vector, should usually not be set manually.
`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)
```

LayoutDbi *Format Log Events for Output to Databases*

Description

LayoutDbi can contain `col_types` that AppenderDbi can use to create new database tables; however, it is safer and more flexible to set up the log table up manually with an `SQL CREATE TABLE` statement instead.

Details

The LayoutDbi parameters `fmt`, `timestamp_fmt`, `colors` and `pad_levels` are only applied for for console output via the `$show()` method and do not influence database inserts in any way. The inserts are pre-processed by the methods `$format_data()`, `$format_colnames` and `$format_tablenames`.

It does not format LogEvents directly, but their `data.table` representations (see as.data.table.LogEvent), as well as column- and table names.

Usage

```
x <- LayoutDbi$new(col_types = NULL, fmt = "%L [%t] %m %f", timestamp_fmt =
  "%Y-%m-%d %H:%M:%S", colors = getOption("lgr.colors", list()), pad_levels =
  "right", format_table_name = identity, format_colnames = identity,
  format_data = identity)

x$clone(deep = FALSE)
x$format_event(event)
x$set_col_types(x)
x$set_colors(x)
x$set_fmt(x)
x$set_pad_levels(x)
x$set_timestamp_fmt(x)
x$sql_create_table(table)
x$toString()

x$col_names
x$col_types
x$colors
x$fmt
x$format_colnames
x$format_data
x$format_table_name
x$pad_levels
x$timestamp_fmt
```

Creating a New Layout

Layouts are instantiated with `<LayoutSubclass>$new()`. For a description of the arguments to this function please refer to the Fields section.

Fields

`col_types` A named `character` vector of column types supported by the target database. If not `NULL` this is used by AppenderDbi or similar Appenders to create a new database table on instantiation of the Appender. If the target database table already exists, `col_types` is not used.

`col_names` Convenience method to get the names of the `col_types` vector

Methods

`format_table_name(x)` Format table names before inserting into the database. For example some databases prefer all lowercase names, some uppercase. SQL updates should be case-agnostic, but sadly in practice not all DBI backends behave consistently in this regard

`format_colnames` Format column names before inserting into the database. See `$format_table_name` for more info

`format_data` Format the input `data.table` before inserting into the database. Usually this function does nothing, but for example for SQLite it has to apply formatting to the timestamp.

`col_names` Convenience method to get the names of the `col_types` vector

`format_event(event)` format a LogEvent

Database Specific Layouts

Different databases have different data types and features. Currently the following `LayoutDbi` subclasses exist that deal with specific databases, but this list is expected to grow as lgr matures:

- `LayoutSqlite`: For SQLite databases
- `LayoutPostgres`: for Postgres databases
- `LayoutMySql`: for MySQL databases
- `LayoutDb2`: for DB2 databases

The utility function `select_dbi_layout()` tries returns the appropriate Layout for a DBI connection, but this does not work for odbc and JDBC connections where you have to specify the layout manually.

See Also

`select_dbi_layout()`, DBI::DBI,

Other Layouts: `LayoutFormat`, `LayoutGlue`, `LayoutJson`, `Layout`

LayoutFormat *Format Log Events as Text*

Description

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.

Usage

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

x$clone(deep = FALSE)
x$format_event(event)
x$set_colors(x)
x$set_fmt(x)
x$set_pad_levels(x)
x$set_timestamp_fmt(x)
x$toString()

x$colors
x$fmt
x$pad_levels
x$timestamp_fmt
```

Creating a New LayoutFormat

A new LayoutFormat is instantiated with `LayoutFormat$new()`. For a description of the arguments to this function please refer to the Fields, and the documentation of `format.LogEvent()`.

Fields

`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 passed on on `format.LogEvent()`

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

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

`%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`

Methods

`format_event(event)` format a LogEvent

See Also

Other Layouts: `LayoutDbi`, `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.

Usage

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

x$clone(deep = FALSE)
x$format_event(event)
x$set_colors(x)
x$set_fmt(x)
x$toString()

x$fmt
```

Creating a New LayoutGlue

A new `LayoutGlue` is instantiated with `LayoutGlue$new()`. It takes a single argument `fmt` that is passed on to `glue::glue()` for each LogEvent.

Fields

`fmt` see `glue::glue()`

Methods

`format_event(event)` format a LogEvent

See Also

lgr exports a number of formatting utility functions that are useful for layout glue: `colorize_levels()`, `pad_left()`, `pad_right()`.

Other Layouts: `LayoutDbi`, `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$set_fmt(
  "{logger$name} {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

Format a LogEvent as JSON

Usage

```
x <- LayoutJson$new(toJSON_args = list(auto_unbox = TRUE))

x$clone(deep = FALSE)
x$format_event(event)
x$set_toJSON_args(x)
x$toString()

x$toJSON_args
```

Creating a New Layout

Layouts are instantiated with `<LayoutSubclass>$new()`. For a description of the arguments to this function please refer to the Fields section.

Fields

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

Methods

`format_event(event)` format a LogEvent

See Also

`read_json_lines()`, `http://jsonlines.org/`

Other Layouts: `LayoutDbi`, `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 *Events - 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 Appenders.

Usage

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

x$clone(deep = FALSE)

x$.logger
x$caller
x$level
x$level_name
x$logger
x$msg
x$timestamp
x$values
```

Creating LogEvents / Fields

The arguments to LogEvent$new() directly translate to the fields stored in the LogEvent:

`level integer`: the log_level / priority of the LogEvent

`timestamp POSIXct` the time when then the LogEvent was created

`caller character`. The name of the calling function

`msg character`. A message

`logger character` scalar. Name of the Logger that created the event (`.logger$full_name`)

`user character` scalar. User as set for the Logger that created this event (`.logger$user`)

`.logger` a Logger. A reference to the Logger that created the event

`...` 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 an Appender that supports custom fields quite naturally.

Usually the above 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.

Active Bindings

LogEvents contain some some active bindings that make it easier to retrieve commonly used values.

`level_name character`: the log_level / priority of the LogEvent labelled according to `getOption("lgr.log_levels")`

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

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

`logger_user character` scalar: The user of the Logger that created this event, equivalent to `event$logger_user`)

See Also

`as.data.frame.LogEvent()`

Examples

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

# The last LogEvent produced by a Logger is stored in the last_event field
lg$last_event  # formatted by default
lg$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 the log message along with metadata (timestamp, calling function) and dispatches it to one or several 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 only necessary if you want to take advantage of hierarchical logging as outlined in `vignette("lgr",package = "lgr")`.

Usage

```
# Cannonical way to initialize a new Logger (see "Creating Loggers")
lg <- get_logger("logger")


# R6 constructor (not recommended for productive use)
lg <- Logger$new(name = "(unnamed logger)", appenders = list(), threshold =
  NULL, filters = list(), exception_handler = default_exception_handler,
  propagate = TRUE)

lg$add_appender(appender, name = NULL)
lg$add_filter(filter, name = NULL)
lg$config(cfg, file, text, list)
lg$debug(msg, ..., caller = get_caller(-8L))
lg$error(msg, ..., caller = get_caller(-8L))
lg$fatal(msg, ..., caller = get_caller(-8L))
lg$filter(event)
lg$handle_exception(...)
lg$info(msg, ..., caller = get_caller(-8L))
lg$log(level, msg, ..., timestamp = Sys.time(), caller = get_caller(-7))
lg$remove_appender(pos)
lg$remove_filter(pos)
lg$set_appenders(x)
lg$set_exception_handler(fun)
lg$set_filters(filters)
lg$set_propagate(x)
lg$set_threshold(level)
lg$spawn(name, ...)
lg$trace(msg, ..., caller = get_caller(-8L))
lg$warn(msg, ..., caller = get_caller(-8L))

lg$ancestry
lg$appenders
lg$exception_handler
lg$filters
lg$inherited_appenders
lg$last_event
lg$name
lg$parent
lg$propagate
lg$threshold
```

Creating Loggers

If you are a package developer you should define a new Logger for each package, but you do not need to configure it. Usually only the root logger needs to be configured (new Appenders added/removed, Layouts modified, etc...).

Loggers should never be instantiated directly with `Logger$new()` but rather via `get_logger("name")`. If `"name"` does not exist, a new Logger with that name will be created, otherwise the function returns a Reference to the existing Logger.

The `name` is potentially a `/` separated hierarchical value like `foo/bar/baz`. Loggers further down the hierarchy are children of the loggers above. (This mechanism does not work of the Logger is initialized with `Logger$new()`)

All calls to `get_logger()` with the same name return the same Logger instance. This means that Logger instances never need to be passed between different parts of an application.

If you just want to log to an additional output (like a log file), you want a new Appender, not a new Logger.

Fields

You can modify the fields of an existing Logger with `logger$set_<fieldname>(value)` (see examples). Another way to configure loggers is via its `$config()` method.

`appenders`, `set_appenders(x)` A 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 structures).

`threshold`, `set_threshold(level)` `character` or `integer` scalar. The minimum log level that triggers this Logger

`exception_handler`, `set_exception_handler()` a `function` that takes a single argument `e`. The function used to handle errors that occur during logging. Defaults to demoting errors to warnings.

`propagate`, `set_propagate()` `TRUE` or `FALSE`. Should LogEvents be passed on to the appenders of the ancestral Loggers?

`filters`, `set_filters(filters)` a `list` that may contain `functions` or any R object with a `filter()` method. These functions must have exactly one argument: `event` which will get passed the LogEvent when the Filterable's `filter()` method is invoked. If all of these functions evaluate to `TRUE` the LogEvent is passed on. Since LogEvents have reference semantics, filters can also be abused to modify them before they are passed on. Look at the source code of `with_log_level()` or `with_log_value()` for examples.

Read-Only Bindings

In addition to the active bindings used to access the fields described above, Loggers also have the following additional read-only bindings:

`name` `character` scalar. A hierarchical value (separated by '"/"') that indicates the loggers name and its ancestors. If a logger is created with `get_logger()` uniqueness of the name is enforced.

`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)`

`inherited_appenders` A `list` of all inherited appenders from ancestral Loggers of the current Logger

`last_event` The last LogEvent produced by the current Logger

Methods

`fatal(msg, ..., caller = get_caller(-8L))` Logs a message with level `fatal` on this logger. If there are *unnamed* arguments in `...`, they will be passed to `base::sprintf()` along with message. Named arguments will be passed as custom fields to LogEvent. If there are named arguments the names must be unique. `caller` refers to the name of the calling function and if specified manually must be a `character` scalar.

`error(msg, ..., caller = get_caller(-8L))` Logs a message with level `error` on this logger. The arguments are interpreted as for `fatal()`.

`warn(msg, ..., caller = get_caller(-8L))` Logs a message with level `warn` on this logger. The arguments are interpreted as for `fatal()`.

`info(msg, ..., caller = get_caller(-8L))` Logs a message with level `info` on this logger. The arguments are interpreted as for `fatal()`.

`debug(msg, ..., caller = get_caller(-8L))` Logs a message with level debug on this logger. The arguments are interpreted as for `fatal()`.

`trace(msg, ..., caller = get_caller(-8L))` Logs a message with level `trace` on this logger. The arguments are interpreted as for `fatal()`.

`log(level, msg, ..., timestamp, caller)` If the `level` passes the Logger `threshold` a new LogEvent with `level`, `msg`, `timestamp` and `caller` is created. Unnamed arguments in `...` will be combined with `msg` via `base::sprintf()`. Named arguments in `...` will be passed on to `LogEvent$new()` as custom fields. If no unnamed arguments are present, `msg` will *not* be passed to `sprintf()`, so in that case you do not have to escape `"%"` characters. If the new LogEvent passes this Loggers filters, it will be dispatched to the relevant Appenders and checked against their thresholds and filters.

`config(cfg, file, text, list` Load a Logger configuration. `cfg` can be either

- 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`,
- `NULL` (to reset the logger config to the default/unconfigured state)

The arguments `file`, `text` and `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.

`add_appender(appender, name = NULL)`, `remove_appender(pos)` Add or remove an Appender. Supplying a `name` is optional but recommended. After adding an Appender with `logger$add_appender(AppenderCons = "console")` you can refer to it via `logger$appenders$console`. `remove_appender()` can remove an Appender by position or name.

`spawn(...)` Spawn a child Logger. `get_logger("foo/bar")$spawn("baz")` is equivalent to `get_logger("foo/bar/baz")`, but can be convenient for programmatic use when the name of the parent Logger is not known.

`filter(event)` Determine whether the LogEvent `x` should be passed on to Appenders (`TRUE`) or not (`FALSE`). See also the active binding `filters`

`add_filter(filter, name = NULL)`, `remove_filter(pos)` Add or remove a filter. When adding a filter an optional `name` can be specified. `remove_filter()` can remove by position or name (if one was specified)

LoggerGlue

`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()`.

`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.

See Also

glue

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)
```

`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 Logger
`threshold`	see Logger
`filters`	see Logger
`exception_handler`	see Logger
`propagate`	see Logger
`x`	any R object. Especially: • A `character` scalar. This can either be the path to a YAML file or a character scalar containing valid YAML • a list containing the elements `appenders`, `threshold`, `exception_handler`, `propagate` and `filters`. See the section *Fields* in Logger for details. • a Logger object, to clone its configuration.

Value

a `list` with the subclass `"logger_config"`

a logger_config object

See Also

https://yaml.org/

---

logger_tree *Logger Tree*

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.

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", "fooooo"), pad = ".")
```

print.Appender *Print an Appender object*

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

x	any R Object
color	TRUE or FALSE: Output with color? Requires the Package **crayon**
...	ignored

Value

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

Examples

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

print.LogEvent *Print or Format Logging Data*

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 LogEvent or lgr_data Object
fmt	A character scalar that may contain any of the tokens listed bellow in the section Format Tokens.
timestamp_fmt	see format.POSIXct()
colors	A list of functions that will be used to color the log levels (likely from crayon::crayon).
log_levels	a named integer vector of 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

`%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 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 **crayon**
...	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 colorized via the package **crayon**?
...	passed on to cli::tree()

Value

x (invisibly)

---

read_json_lines *Read a JSON logfile*

Description

Read a JSON logfile

Usage

```
read_json_lines(file)
```

Arguments

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

Value

a data.frame

See Also

LayoutJson

select_dbi_layout *Select Appropriate Database Table Layout*

Description

Selects an appropriate Layout for a database table based on a DBI connection and - if it already exists in the database - the table itself.

Usage

```
select_dbi_layout(conn, table)
```

Arguments

conn	a DBI connection
table	a character scalar. The name of the table to log to.

simple_logging *Simple Logging*

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 lgr$info(), lgr$debug(), etc... .
caller	a character scalar. The name of the calling function
level	integer or character scalar: the desired log level
target	a Logger or Appender 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 log level 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

```
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()
```

---

suspend_logging *Suspend All Logging*

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")
```

---

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 Logger or the name of one (see `get_logger()`). 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