Package 'logtree'

August 6, 2026

Title Tree-Style Console Logger for Nested Processes

Version 0.1.0

Description Render nested process execution as a live, colored tree in the console, with tree connectors, status glyphs, and elapsed time per step. Nesting depth is tracked via frame exit handlers so it never desynchronizes, even when a step errors. Builds on the 'cli' package for console rendering.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 8.0.0

Depends R (>= 4.0)

Imports cli, rlang, withr

Suggests covr, jsonlite, knitr, logger, pkgdown, rmarkdown, testthat (>= 3.1.4)

Config/testthat/edition 3

VignetteBuilder knitr

URL https://github.com/IvanSortino/logtree, https://ivansortino.github.io/logtree/

BugReports https://github.com/IvanSortino/logtree/issues

NeedsCompilation no

Author Ivan Sortino [aut, cre, cph]

Maintainer Ivan Sortino <ivan.sortino97@gmail.com>

Repository CRAN

Date/Publication 2026-08-06 13:40:08 UTC

Contents

`layout_logtree` *A* `logger` *layout that renders through logtree*

Description

Bridges the `logger` package (https://daroczig.github.io/logger/) into `logtree`'s tree rendering. `logger`'s own per-call pipeline is `formatter()` -> `layout()` -> `appender()`: only the *layout* stage receives the structured `level` object (an integer with a `"level"` attribute such as `"INFO"`) – `appender()` only ever sees a pre-formatted character line – so a custom layout, not a custom appender, is the correct integration point. Register it as `logger`'s layout and pair it with `logger::appender_void` (a ready-made no-op) so that `logtree`'s rendering, which happens as a side effect of the layout call, is the only visible output:

Usage

```
layout_logtree(
  level,
  msg,
  namespace = NA_character_,
  .logcall = sys.call(),
  .topcall = sys.call(-1),
  .topenv = parent.frame(),
  .timestamp = Sys.time()
)
```

Arguments

`level`	A `logger` log level object (e.g. `logger::INFO`), as passed in by `logger`'s internal dispatch.
`msg`	Character scalar, already formatted by `logger`'s formatter stage (glue interpolation has already happened by this point).

namespace, .logcall, .topcall, .topenv, .timestamp
: Unused; accepted only because logger's dispatcher calls every layout with this exact signature (see logger::layout_simple).

Details

```
logger::log_layout(logtree::layout_logtree)
logger::log_appender(logger::appender_void)
```

logger severities map onto logtree leaf levels as: FATAL/ERROR -> log_error(), WARN -> log_warn(), SUCCESS -> log_success(), INFO -> log_info(), DEBUG/TRACE -> log_debug() (logger has two debug-ish tiers, logtree has one, so both collapse to the same leaf). Note logger's own log_threshold() already gates before the layout is ever invoked; logtree_threshold() is then an independent, second gate applied on top of that – both legitimately apply at once, this is not a bug.

Value

character(0), invisibly. The record is discarded by logger::appender_void() regardless, so its content is irrelevant; a zero-length character vector matches logger's layout contract.

Examples

```
if (requireNamespace("logger", quietly = TRUE)) {
  logtree_reset()
  logger::log_layout(layout_logtree, namespace = "logtree_demo")
  logger::log_appender(logger::appender_void, namespace = "logtree_demo")
  log_step("Demo step")
  logger::log_info("hello", namespace = "logtree_demo")
}
```

---

logtree_logger *Route the* logger *package through logtree*

Description

Call once near the top of a script to make the logger package (https://daroczig.github.io/logger/) render through logtree. It registers layout_logtree() as logger's layout and logger::appender_void as its appender for namespace, so from then on every logger::log_info() / log_warn() / ... call in that namespace prints as a logtree leaf. This is the one-call form of the manual logger::log_layout() + logger::log_appender() pairing.

Usage

```
logtree_logger(namespace = "global", threshold = TRUE)
```

Arguments

namespace	logger namespace to route. Default "global", the namespace a bare logger::log_info("x") uses.
threshold	Open logger's threshold to TRACE for namespace so logtree_threshold() is the only gate? Default TRUE.

Details

With threshold = TRUE (the default) it also opens logger's own threshold to TRACE for the namespace. logger gates on its threshold *before* the layout runs, so without this a logger::log_debug() would never reach logtree; opening it makes logtree_threshold() the single effective gate.

Bridge only: it does not install error handling. Wrap the run body in with_logging() as well when you want failed-run elevation and a summary line. The change is persistent for the session (matching logger's own global configuration style); there is no automatic teardown.

Value

NULL, invisibly.

See Also

layout_logtree() for the underlying layout, with_logging() for top-level error handling.

Examples

```
if (requireNamespace("logger", quietly = TRUE)) {
  logtree_reset()
  logtree_logger(namespace = "logtree_demo")
  log_step("Demo step")
  logger::log_info("hello", namespace = "logtree_demo")
}
```

---

logtree_reset *Reset internal logtree state*

Description

Clears the open-step stack and resets the internal id counter. Mainly useful for tests and interactive/knitr re-runs where a previous run may have left the stack non-empty (e.g. after an uncaught error with no with_logging() wrapper).

Usage

```
logtree_reset()
```

Value

NULL, invisibly.

Examples

```
logtree_reset()
```

`logtree_sink_file` *Add a file sink*

Description

Registers an additional output destination. Every logged event fans out to the console sink (always on) and every registered file sink, so console, text-file, and NDJSON outputs can all run simultaneously (design doc section 6).

Usage

```
logtree_sink_file(path, format = c("text", "json"))
```

Arguments

`path`	File path to append rendered log lines to.
`format`	`"text"` for a plain ASCII tree (no ANSI, independent of the active console theme) or `"json"` for one NDJSON object per event.

Value

`NULL`, invisibly.

Examples

```
logtree_reset()
logtree_sink_file(tempfile(), format = "text")
with_logging({
  log_step("Step one")
})
```

`logtree_summary` *Report a digest of notable events*

Description

Prints a compact end-of-run digest of everything worth attention that happened since the last `logtree_reset()`: every warning and error leaf line, plus any step that closed with a `warning`, `error`, or `interrupted` status. Each entry shows the status glyph, the breadcrumb path to where it happened, and the message (for leaf lines) or an outcome word (for steps).

Usage

```
logtree_summary(filter = NULL, depth = NULL)
```

Arguments

filter	Optional character vector of statuses to include, e.g. "error" or c("warning", "interrupted"). Only entries whose status matches are printed and returned; recognised statuses are "error", "warning", "interrupted", and the pinned leaf statuses "info", "success", "debug". NULL (the default) reports every entry.
depth	Optional positive integer limiting how many trailing (deepest) breadcrumb nodes are printed. The message counts as the terminal node, so depth = 1 prints just the message (or, for a step entry, its innermost step), depth = 2 the message plus its immediate parent, and so on. NULL (the default) prints the full breadcrumb. Affects printing only; the returned entries always carry the full path.

Details

Unlike scrolling the live tree, the digest surfaces breakage even when no with_logging() handler was installed – interrupted steps are picked up from their close lines. Ordinary info / success lines are excluded unless logged with summary = TRUE; a warning or error can be excluded with summary = FALSE.

Value

The recorded entries, invisibly: a list of records, each a list with kind, status, msg, path (character vector), and elapsed.

See Also

with_logging(), logtree_reset()

Examples

```
logtree_reset()
f <- function() {
  log_step("Load data")
  log_warn("coerced 3 rows")
}
f()
logtree_summary()
```

`logtree_theme` *Set the active glyph/color theme*

Description

Set the active glyph/color theme

Usage

```
logtree_theme(
  theme = c("unicode", "ascii", "emoji"),
  overrides = list(),
  compact = FALSE
)
```

Arguments

`theme`	Either a preset name (`"unicode"`, `"ascii"`, `"emoji"`) to swap the whole glyph set, or a named list of per-key overrides to merge onto the currently active theme (matching the two calling styles shown in the package documentation).
`overrides`	A named list of per-key overrides applied on top of `theme` after it is resolved. Each entry may specify `glyph`, `width`, and/or `color`; unspecified fields are kept from the existing entry. The `group` slot also accepts `bracket` (logical, default `FALSE`): when `TRUE` the header name is wrapped in `< >`.
`compact`	Density of the tree's per-level indentation. `FALSE` (the default) keeps the normal spacing (three columns per level in the unicode theme); `"medium"` drops the trailing gap after each connector (two columns per level); `"tight"` additionally slims the branch and corner connectors to a single character (one column per level). `TRUE` is an alias for `"tight"`. Compact applies to the active (console) theme and is cleared by a subsequent preset swap such as `logtree_theme("unicode")`.

Details

An override list is keyed by *slot*; each slot's value is itself a named list of *fields*. Only the fields you name are changed – everything else is kept from the active theme.

Slots (valid names in an override / preset list):

Slot	Applies to	Fields it accepts
`step`	open / running step glyph	`glyph`, `width`, `color`
`info`	`log_info()` leaf	`glyph`, `width`, `color`
`debug`	`log_debug()` leaf	`glyph`, `width`, `color`
`success`	success glyph (clean close, `log_success()`)	`glyph`, `width`, `color`
`warning`	`log_warn()` / elevated step glyph	`glyph`, `width`, `color`
`error`	`log_error()` / elevated step glyph	`glyph`, `width`, `color`
`interrupted`	abnormal-exit (dimmed) glyph	`glyph`, `width`, `color`
`group`	group header marker	`glyph`, `color`, `bracket`

branch	child connector: the "tee" drawn before every child line	glyph, color
corner	close-line connector: the "elbow" drawn on a step's own close line	glyph, color
pipe	vertical rail carried down the left of nested lines	glyph, color

Fields (valid names inside a slot):

Field	Type	Accepted values
glyph	character(1)	Any string, including "". In package source, non-ASCII must be written as \u/\U escapes, n
width	integer(1)	Rendered display width of glyph (1 for normal, 2 for emoji / wide cells). Drives column alig
color	character or NULL	One or more cli styles, or NULL for no styling. Named colors ("red", "cyan", "silver", ...)
bracket	logical(1)	group slot only. TRUE wraps the header name in < >; default FALSE.

Value

NULL, invisibly.

Examples

```
logtree_theme("ascii")
logtree_theme("unicode")
logtree_theme(overrides = list(success = list(glyph = "*")))
logtree_theme(overrides = list(group = list(glyph = "#", bracket = TRUE)))
logtree_theme("unicode", compact = "medium")
logtree_theme("unicode", compact = "tight")
logtree_theme("unicode")
```

---

logtree_threshold *Set the minimum log level threshold to render*

Description

Leaf lines below this level are silently skipped: log_debug() counts as "debug", log_info() and log_success() count as "info", log_warn() as "warn", log_error() as "error". Step open/close lines always render regardless of verbosity, since hiding them would break the tree structure. Suppressed log_warn()/log_error() calls still elevate the enclosing step's close glyph – verbosity only hides the leaf line's own text.

Usage

```
logtree_threshold(level = c("debug", "info", "warn", "error"))
```

Arguments

level	One of "debug", "info", "warn", "error" (case-insensitive).

Value

NULL, invisibly.

Examples

```
logtree_threshold("info")
```

---

log_close *Close a manually-opened step*

Description

Closes the step opened by log_open() with the given id, cascading to any of its still-open descendants (deepest-first). With no id, closes the nearest open step, so simple last-in-first-out use needs no handle at all.

Usage

```
log_close(id = NULL, status = NULL)
```

Arguments

id	Step handle from log_open(). If omitted, the nearest open step is closed.
status	Optional character scalar overriding the step's final status: one of "success", "warning", or "error". Bypasses the usual elevation rule instead of comparing against it.

Details

A step's status only ever escalates via log_warn()/log_error() (see status elevation); it never comes back down on its own, so a step that logged an error and then recovered still closes with the error glyph. Pass status to override that explicitly – this force-assigns the step's final status regardless of what it escalated to. Because id = NULL resolves to the nearest open step for both log_open()-managed and log_step()-managed steps alike, this also lets you close (and override) a log_step() step early, before its automatic close-on-frame-exit fires.

Value

A list with status and elapsed (seconds) for the step just closed, invisibly – the same values rendered on its Done line ("running" resolves to "success", as it does for display). NULL, invisibly, if there was no open step to close.

See Also

log_open()

Examples

```
logtree_reset()
log_open("Step 1")
log_info("a child line")
log_close()

logtree_reset()
log_open("Step 2")
log_error("failed once")
log_close(status = "success")  # recovered: override the elevated glyph

logtree_reset()
log_open("Step 3")
result <- log_close()  # result$status, result$elapsed
```

---

log_debug *Log a debug leaf line*

Description

The most verbose leaf level, for fine-grained diagnostic detail that would be noisy at the default verbosity. Shown only when verbosity is "debug" (see logtree_threshold()). Like log_info() and log_success(), it does not elevate the enclosing step's status – unlike log_warn()/log_error().

Usage

```
log_debug(msg, close = FALSE, summary = NA)
```

Arguments

msg	Character scalar.
close	Logical. When TRUE, force-close the enclosing section after this line: the line is shown as the section's terminal (corner) line and its Done line is suppressed. Defaults to FALSE.
summary	Whether to record this line in the logtree_summary() digest. NA (default) records it only when it is a warning or error; TRUE always records it; FALSE never does.

Value

NULL, invisibly.

Examples

```
logtree_reset()
logtree_threshold("debug")
log_debug("Cache miss for key user:42")
logtree_threshold("info")
```

log_error *Log an error leaf line*

Description

Also elevates the currently-open step's status to `"error"`, so the step's close line renders the elevated glyph even though the enclosing function returns normally (see `with_logging()` for the case where the step's code actually throws instead).

Usage

```
log_error(msg, close = FALSE, summary = NA)
```

Arguments

`msg`	Character scalar.
`close`	Logical. When `TRUE`, force-close the enclosing section after this line: the line is shown as the section's terminal (corner) line and its `Done` line is suppressed. Defaults to `FALSE`.
`summary`	Whether to record this line in the `logtree_summary()` digest. `NA` (default) records it only when it is a warning or error; `TRUE` always records it; `FALSE` never does.

Value

`NULL`, invisibly.

Examples

```
logtree_reset()
log_error("model timeout after 30s")
```

log_info *Log an informational leaf line*

Description

Log an informational leaf line

Usage

```
log_info(msg, close = FALSE, summary = NA)
```

Arguments

msg	Character scalar.
close	Logical. When TRUE, force-close the enclosing section after this line: the line is shown as the section's terminal (corner) line and its Done line is suppressed. Defaults to FALSE.
summary	Whether to record this line in the logtree_summary() digest. NA (default) records it only when it is a warning or error; TRUE always records it; FALSE never does.

Value

NULL, invisibly.

Examples

```
logtree_reset()
log_info("Reading config.yml")
```

---

log_open *Open a step under manual lifetime control*

Description

Like log_step() but with no automatic close: the step stays open until you close it yourself with log_close(). This is what you want at top level (a script or the REPL), where there is no enclosing function frame for log_step() to hang its close on. You may also attach the step to a chosen open parent rather than the innermost open step, letting you build the tree by hand.

Usage

```
log_open(
  msg,
  glyph = NULL,
  parent = NULL,
  group = NULL,
  close = FALSE,
  key = NULL
)
```

Arguments

msg	Character scalar. The step's label.
glyph	Optional character scalar overriding this step's glyph.
parent	Optional step handle (an id returned by log_open()/ log_step()) of a currently-open step to nest this one under. Defaults to the innermost open step. The target must still be open, else an error.

group	Optional named length-1 vector c(name = value), as in log_step().
close	Logical. When TRUE, the step is force-closed silently as soon as its opening line is printed: a header-only marker with no children and no Done line. Defaults to FALSE.
key	Optional character scalar giving this step a stable identity for re-run reconciliation, as in log_step(). At top level the label is used automatically, so re-running the same log_open() line re-anchors to that node instead of nesting under the previous run's leftovers. Ignored when parent is supplied.

Details

Opening a step at the same depth as an already-open step – for example by linking to a shared parent – first closes that sibling and its descendants, since a new sibling means the previous subtree is done.

Value

The step's id, invisibly. Capture it to pass to log_close() or as another step's parent.

See Also

log_close(), log_step()

Examples

```
logtree_reset()
s1 <- log_open("Step 1")
log_info("a child line")
log_close(s1)
```

---

log_step *Open a logged step*

Description

log_step() is intended to be called from *inside a function*: it prints an opening line for msg and registers an automatic close that fires when the *calling* function's frame exits – whether by normal return, early return(), or an uncaught error propagating through it. Because the close is registered in the caller's frame rather than inside log_step() itself, nesting depth always stays in sync, even across errors. At top level, where there is no enclosing function frame to close on, use log_open() / log_close() instead.

Usage

```
log_step(
  msg,
  glyph = NULL,
  parent = NULL,
  group = NULL,
  close = FALSE,
  key = NULL
)
```

Arguments

msg	Character scalar. The step's label.
glyph	Optional character scalar overriding this step's glyph.
parent	Optional step handle (an id returned by log_open()/ log_step()) of a currently-open step to nest this one under. Defaults to the innermost open step. The target must still be open, else an error.
group	Optional named length-1 vector c(name = value). Adjacent log_step() calls sharing the same value are grouped under a single < name > header line. The value is the match key; the name is displayed.
close	Logical. When TRUE, the step is force-closed silently as soon as its opening line is printed: a header-only marker with no children and no Done line (and no automatic close is registered). Defaults to FALSE.
key	Optional character scalar giving this step a stable identity for re-run reconciliation. At top level (the global env) the label is used automatically, so re-running the same line re-anchors to that node instead of nesting under the previous run's leftovers. Pass key to override the automatic label key, or to keep two same-label steps that are open at once distinct. Ignored when parent is supplied.

Details

Opening a step at the same depth as an already-open step retires that earlier sibling automatically – its close line is printed with no explicit log_close() call. In the default nested pattern each log_step() descends one level deeper, so this same-level retirement applies when you place steps side by side via an explicit parent.

Value

The step's internal id, invisibly.

Examples

```
logtree_reset()
f <- function() {
  log_step("Doing work")
}
f()
```

log_success *Log a success leaf line*

Description

Log a success leaf line

Usage

```
log_success(msg, close = FALSE, summary = NA)
```

Arguments

msg	Character scalar.
close	Logical. When TRUE, force-close the enclosing section after this line: the line is shown as the section's terminal (corner) line and its Done line is suppressed. Defaults to FALSE.
summary	Whether to record this line in the logtree_summary() digest. NA (default) records it only when it is a warning or error; TRUE always records it; FALSE never does.

Value

NULL, invisibly.

Examples

```
logtree_reset()
log_success("Validated 12 parameters")
```

log_warn *Log a warning leaf line*

Description

Also elevates the currently-open step's status to "warning" (unless it is already "error"), so the step's close line renders the elevated glyph even though the enclosing function returns normally.

Usage

```
log_warn(msg, close = FALSE, summary = NA)
```

Arguments

msg	Character scalar.
close	Logical. When TRUE, force-close the enclosing section after this line: the line is shown as the section's terminal (corner) line and its Done line is suppressed. Defaults to FALSE.
summary	Whether to record this line in the logtree_summary() digest. NA (default) records it only when it is a warning or error; TRUE always records it; FALSE never does.

Value

NULL, invisibly.

Examples

```
logtree_reset()
log_warn("Retry 1/3 due to timeout")
```

---

with_logging *Run an expression with top-level error handling and a run summary*

Description

Wrap a script or pipeline's top-level call in with_logging() so an uncaught error leaves a clean, correctly-colored tree instead of dimmed "interrupted" steps. On error, every currently open step is marked failed, the error is logged as a leaf line, then rethrown – with_logging() never silently swallows errors. It also prints a "Run complete" / "Run failed" summary line with elapsed time.

Usage

```
with_logging(expr, summary = TRUE, global = FALSE)
```

Arguments

expr	Code to run. Omitted when global = TRUE.
summary	Print an end-of-run summary line? Default TRUE. In global mode only the "Run failed" line is printed (on an uncaught error); there is no frame exit to hang a "Run complete" line on.
global	If TRUE, do not wrap an expression; instead install a session-persistent global error handler for use at the *top level of a script*. On an error that reaches top level *unhandled* while logtree steps are open, it marks those steps failed and logs the error message as a leaf – the same result as the block form, but without wrapping the body. It fires only for genuinely uncaught top-level errors (an inner tryCatch() that catches first pre-empts it) and only when steps are open. The handler persists until logtree_reset(). Must be called from a clean top level – the first line of a script: calling it where condition handlers are active (inside tryCatch(), or a function running under one) errors, by design. Requires R (>= 4.0).

Details

Note: `expr` is lazily evaluated, so `log_step()` calls written inside the `{ ... }` block close when the function *lexically enclosing* that block returns – not necessarily when `with_logging()` itself returns. Use `with_logging({ ... })` as a function's entire body to keep these in sync; if other code runs after the call in the same function, steps opened inside the block stay open until that function returns.

The `global = TRUE` form is meant for the top level of a script, where there is no frame to wrap. It is not shown in the examples below because it installs a session-persistent handler and is only meaningful for an error that reaches top level:

```
with_logging(global = TRUE)
log_open("Load data")
stop("EOF")   # marks the open step failed + logs "EOF" before R exits
```

Value

In block mode, the value of `expr`, invisibly. In global mode, `NULL`, invisibly.

Examples

```
logtree_reset()
with_logging({
  log_step("Step one")
  log_success("done")
})
```

Index