Package 'logtree'

September 1, 2026

Title Tree-Style Console Logger for Nested Processes

Version 0.2.0

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

Depends R (>= 4.0)

Imports cli, rlang, withr

Suggests covr, jsonlite, knitr, logger (>= 0.3.0), 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

Config/roxygen2/version 8.1.0

NeedsCompilation no

Author Ivan Sortino [aut, cre, cph]

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

Repository CRAN

Date/Publication 2026-09-01 10:30:10 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`, `.timestamp`	Unused; accepted only because `logger`'s dispatcher calls every layout with this exact signature (see `logger::layout_simple`).
`.topcall`	The call `logger` was invoked from, supplied by its dispatcher. Used as the leaf's call site for the `trace` column.
`.topenv`	Unused; part of `logger`'s layout signature.

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.

When the `trace` theme slot is on, the leaf's call site is taken from `logger`'s own `.topcall` rather than from logtree's usual frame walk. That matters: this layout is what calls the leaf, so a frame walk would report `layout_logtree` as the origin of every routed line instead of the `logger::log_info()` call in your code.

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 (rlang::is_installed("logger", version = "0.3.0")) {
  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.

Needs `logger` >= 0.3.0, the release that added `appender_void` – the no-op appender this pairs the layout with, so that logtree's rendering is the only visible output. An older `logger` is refused with a message saying so rather than failing later on a missing object.

Value

`NULL`, invisibly.

See Also

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

Examples

```
if (rlang::is_installed("logger", version = "0.3.0")) {
  logtree_reset()
  logtree_logger(namespace = "logtree_demo")
  log_step("Demo step")
  logger::log_info("hello", namespace = "logtree_demo")
}
```

`logtree_mute` *Silence logtree's output*

Description

Stops every sink – console, files, and any of your own – from receiving events, without unregistering them. This is what a library that logs with logtree reaches for to keep its own test suite quiet, and what a script reaches for around a noisy section.

Usage

```
logtree_mute()

logtree_unmute()
```

Details

Three things muting deliberately does *not* do:

- It does not stop the run being *recorded*. Warnings and errors still reach the `logtree_summary()` digest, so a muted run can still be asked what went wrong at the end – and `logtree_summary()` still prints when you call it, since that output is one you asked for rather than one that happened to you. The `with_logging()` run-summary line, which you did not ask for, is silenced.
- It does not disturb step bookkeeping. Steps still open and close while muted, so depth is still correct when output comes back.
- It does not survive... anything, actually: like registered sinks, and unlike the open-step stack, the mute flag is left alone by `logtree_reset()`. Unmute explicitly.

The `logtree.silent` option sets the initial state when the package is loaded, for silencing logtree from an `.Rprofile` or a test setup file; after that these functions are in charge, so a later `options(logtree.silent = )` has no effect.

Value

The previous muted state (`TRUE`/`FALSE`), invisibly, so a caller can restore whatever it found.

See Also

`logtree_sink_remove()` for taking a sink off altogether.

Examples

```
logtree_reset()
was <- logtree_mute()

f <- function() {
  log_step("Load data")
  log_warn("coerced 3 rows")
}
invisible(f())          # prints nothing

logtree_unmute()
# ... but the run was still recorded:
length(logtree_summary())

# Restore whatever was in force before, rather than assuming it was off.
was <- logtree_mute()
if (!was) logtree_unmute()
```

`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()
```

Details

Registered sinks are deliberately *not* cleared: they are configuration rather than run state, so a file sink set up once keeps recording across resets. Take one off with `logtree_sink_remove()`.

Value

`NULL`, invisibly.

Examples

```
logtree_reset()
```

`logtree_sink` *Register a sink of your own*

Description

Adds an arbitrary function as an output destination. Every logged event fans out to each registered sink in registration order, so a custom sink runs alongside the console and any file sinks. Use it to push events at a database, a metrics collector, or an in-test collector of your own; for the built-in file destinations see `logtree_sink_file()`.

Usage

```
logtree_sink(fn, threshold = NULL)
```

Arguments

`fn`	A function of one argument, called with each emitted event. Its return value is ignored.
`threshold`	Minimum leaf level this sink receives: one of `"debug"`, `"info"`, `"warn"`, `"error"`. `NULL` (the default) follows the global `logtree_threshold()`, read afresh for each event. Step open/close lines are never gated, whatever the threshold.

Details

`fn` is called with one argument, the event: a list whose `kind` is one of `"open"`, `"close"`, `"group"`, `"group_close"` or `"leaf"`. Step-shaped events (everything but `"leaf"`) carry the step's record under `entry`, while a leaf carries its own fields directly.

A sink that throws does not take the rest of the fanout down with it: the error is caught, the remaining sinks still run, and a warning naming the sink is raised once (per sink, until the next `logtree_reset()`). A logger should witness failures, not become a source of them.

Value

The sink's id, invisibly – pass it to `logtree_sink_remove()`.

See Also

`logtree_sinks()`, `logtree_sink_remove()`, `logtree_sink_file()`, `logtree_sink_memory()`

Examples

```
logtree_reset()
seen <- character(0)
h <- logtree_sink(function(event) seen <<- c(seen, event$kind))

f <- function() log_step("Step one")
invisible(f())
seen
```

```
logtree_sink_remove(h)

# A sink that only ever hears about failures.
h <- logtree_sink(function(event) invisible(NULL), threshold = "error")
logtree_sink_remove(h)
```

---

logtree_sinks *List the registered sinks*

Description

The ids of every sink currently registered, in the order they fire. The console sink is always first under the reserved id `"console"` unless it has been removed.

Usage

```
logtree_sinks()
```

Value

A character vector of sink ids.

See Also

`logtree_sink()`, `logtree_sink_remove()`

Examples

```
logtree_sinks()
```

---

logtree_sink_file *Add a file sink*

Description

Registers an additional output destination. Every logged event fans out to the console sink 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"),
  trace = NULL,
  timestamp = NULL,
  threshold = NULL
)
```

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.
trace	Whether this sink prints the call-site column (see the trace slot in logtree_theme()). NULL (the default) follows the active console theme, read afresh for each event, so switching trace on reaches the file too; FALSE, TRUE or "problems" pins this sink independently of the console. Text sinks only: a "json" sink always carries the fn, file and line fields, null when there is no call site to report.
timestamp	Whether this sink prints the wall-clock column (see the timestamp slot in logtree_theme()). NULL (the default) follows the active console theme, read afresh for each event; FALSE pins it off; TRUE turns it on with a date-and-time format suited to a file that outlives the session ("%Y-%m-%d %H:%M:%S"); a strftime format string pins that format. Text sinks only: a "json" sink always carries ts.
threshold	Minimum leaf level this file records: one of "debug", "info", "warn", "error". NULL (the default) follows the global logtree_threshold(). This is per sink, so a "debug" file can record everything while the console stays at "info" – or an "error" file can keep only what went wrong. Step open/close lines are never gated.

Details

A "json" sink writes one NDJSON object per event, with these fields:

Field	What it holds
ts	when the event was emitted, ISO-8601 to the millisecond with UTC offset
run_id	identifies the run, so one run's lines can be picked out of a shared file
level	event kind: "open", "close", "group", "group_close", "leaf"
id, parent_id, depth	the node's identity and place in the tree
label	a step's label, a group's name, or a leaf's message
elapsed	seconds, on close lines only; null elsewhere
status	a leaf's status, "step"/"group" on an opening line, or the resolved status on a close
fn, file, line	the call site, when the trace theme slot recorded one; null otherwise

Value

The sink's id, invisibly – pass it to logtree_sink_remove() to stop writing to this file.

See Also

logtree_sink() for a sink of your own, logtree_sinks() and logtree_sink_remove() for the registry.

Examples

```
logtree_reset()
logtree_sink_file(tempfile(), format = "text")
```

```
with_logging({
  log_step("Step one")
})

# A file that records call sites even with the console column off.
logtree_sink_file(tempfile(), format = "text", trace = TRUE)

# A debug-level record on disk while the console stays at "info".
logtree_sink_file(tempfile(), format = "json", threshold = "debug")

# A file that stamps every line with the date and time.
logtree_sink_file(tempfile(), format = "text", timestamp = TRUE)
```

`logtree_sink_memory` *Collect logged events in memory*

Description

Registers a sink that keeps every event in a buffer instead of writing it anywhere, so a run's logging can be *asserted on* rather than eyeballed. Read the buffer back with `logtree_sink_memory_events()`. This is the answer to "did my pipeline log what it should have?" – previously that meant capturing console output and pattern-matching the rendered tree.

Usage

```
logtree_sink_memory(max = 1000, threshold = NULL)
```

Arguments

`max`	Maximum number of events to keep. Once the buffer is full the oldest events are dropped, so what you read back is always the most recent `max`. Default `1000`.
`threshold`	Minimum leaf level to collect, as in `logtree_sink()`. `NULL` (the default) follows the global `logtree_threshold()`; pass `"debug"` to collect everything a run logged regardless of what the console was set to show.

Details

The buffer is dropped when the sink is removed with `logtree_sink_remove()`, and it is capped: a long-running process cannot grow it without bound.

Value

The sink's id, invisibly – pass it to `logtree_sink_memory_events()` to read the buffer, and to `logtree_sink_remove()` to stop collecting.

See Also

`logtree_sink_memory_events()`, `logtree_sink()`

Examples

```
logtree_reset()
h <- logtree_sink_memory()

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

events <- logtree_sink_memory_events(h)
events[, c("level", "label", "status")]

logtree_sink_remove(h)
```

`logtree_sink_memory_events` *Read a memory sink's collected events*

Description

Returns everything a `logtree_sink_memory()` sink has collected so far, as a data frame with one row per event and the same columns a `"json"` file sink writes (see `logtree_sink_file()`), so the two views of a run agree:

Usage

```
logtree_sink_memory_events(id)
```

Arguments

`id`	A memory sink's id, as returned by `logtree_sink_memory()`.

Details

Column	What it holds
`ts`	when the event was emitted (`POSIXct`)
`level`	event kind: `"open"`, `"close"`, `"group"`, `"group_close"`, `"leaf"`
`id`, `parent_id`, `depth`	the node's identity and place in the tree
`label`	a step's label, a group's name, or a leaf's message
`elapsed`	seconds, on close lines only; `NA` elsewhere
`status`	a leaf's status, `"step"`/`"group"` on an opening line, or the resolved status on a close
`fn`, `file`, `line`	the call site, when the `trace` theme slot recorded one; `NA` otherwise

Value

A data frame with one row per collected event, oldest first, and zero rows (with the columns above) when nothing has been logged yet.

See Also

logtree_sink_memory()

Examples

```
logtree_reset()
h <- logtree_sink_memory()
f <- function() log_info("Reading config.yml")
invisible(f())
logtree_sink_memory_events(h)
logtree_sink_remove(h)
```

---

logtree_sink_remove *Remove registered sinks*

Description

Unregisters one or more sinks by id. Ids that are not registered are ignored, so cleanup code can run unconditionally. The reserved `"console"` id can be removed like any other, which is how a library silences logtree's console output outright.

Usage

```
logtree_sink_remove(id)
```

Arguments

id	Character vector of sink ids, as returned by logtree_sink(), logtree_sink_file() or logtree_sinks().

Details

Sinks deliberately survive logtree_reset(), so this is the only way to take one off again.

Value

The removed sink functions, invisibly: a named list keyed by the ids actually removed (empty when none matched). Re-registering one with logtree_sink() restores it, under a fresh id and therefore at the end of the firing order.

See Also

logtree_sink(), logtree_sinks()

Examples

```
logtree_reset()
h <- logtree_sink(function(event) invisible(NULL))
logtree_sinks()
logtree_sink_remove(h)
logtree_sinks()
```

`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,
  gap = NULL,
  rule = NULL,
  trace = 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`.
`gap`	Number of blank lines printed between the last log line and the digest; `0` prints the digest flush against the tree. `NULL` (the default) takes the active theme's `summary$gap` (`1` in every built-in preset).
`rule`	Divider drawn above the digest. `TRUE` draws a `cli::rule()` labelled with the digest header, so the counts become the rule's title instead of a separate line; `FALSE` draws no rule and keeps the plain header line; a character string draws the rule with that title and prints the header line below it. `NULL` (the default) takes the active theme's `summary$rule` (`TRUE` in every built-in preset).

`trace` Pins the digest's call-site column for this call, overriding the theme's `trace$show`. Takes the same values: `FALSE` for no call sites, `TRUE` for all of them, `"problems"`, or a vector of statuses such as `"error"` – see `logtree_theme()`. `NULL` (the default) follows the theme, so the digest agrees with the tree. Useful when the tree was quiet and the digest is where you want the locations, or the reverse. Note this can only narrow or reshape what was *captured*: capture is decided while the run happens, so with the theme's slot off for the run there is nothing for `trace = TRUE` here to print. To get the quiet-tree-annotated-digest combination, ask for capture during the run and print nothing: `logtree_theme(list(trace = list(show = FALSE, capture = TRUE)))`, then `logtree_summary(trace = TRUE)`.

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`.

The digest's appearance comes from the active theme, so it is customised through `logtree_theme()` like everything else: the `crumb` slot sets the breadcrumb separator and the emphasis on the path nodes, the `summary` slot the divider (`gap`, `rule`, `line`). `gap`, `rule` and `trace` below override the theme for a single call.

Value

The recorded entries, invisibly: a list of records, each a list with `kind`, `status`, `msg`, `path` (character vector), `elapsed`, and `trace` (the call site: a list of `fn`, `file` and `line`, or `NULL` when the `trace` theme slot was off and nothing was captured).

See Also

`with_logging()`, `logtree_reset()`

Examples

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

# Flush against the tree, with a titled divider.
logtree_summary(gap = 0, rule = "Run report")

# Call sites in the digest only: the tree above stays as it was rendered.
logtree_theme(list(trace = list(show = TRUE)))
f()
logtree_summary(trace = "error")
logtree_theme("unicode")
```

```
# Set the layout and the breadcrumb symbol once, on the theme.
logtree_theme(list(
  summary = list(gap = 2, rule = FALSE),
  crumb   = list(glyph = " / ")
))
logtree_summary()
logtree_theme("unicode")
```

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

Description

Set the active glyph/color theme

Usage

```
logtree_theme(
  theme = NULL,
  overrides = list(),
  compact = FALSE,
  glyph_gap = NULL,
  connector_gap = NULL,
  wrap = NULL
)
```

Arguments

`theme` Either a preset name 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). `NULL` (the default) keeps the active preset: **a preset is swapped only when you name one**, so a call that sets just `overrides`, `compact`, `glyph_gap`, `connector_gap` or `wrap` merges onto whatever theme is active. Reset with `logtree_theme("unicode")`. Five presets ship with the package:

Preset	What it is for
`"unicode"`	The default. Box-drawing connectors and coloured symbol glyphs, for an interactive terminal.
`"ascii"`	Plain ASCII, no colour. Safe for log files, CI, and non-UTF-8 terminals; also what every file sink renders throug
`"emoji"`	Emoji status glyphs (width-2 cells) over box-drawing connectors.
`"minimal"`	No connectors at all: `branch`, `corner` and `pipe` are empty, so depth is carried by indentation alone (two column
`"ci"`	Bracketed word glyphs (`[step]`, `[ok]`, `[warn]`, `[fail]`, ...) over pure-ASCII connectors, with no colour in any

`overrides` A named list of per-slot overrides applied on top of `theme` once it is resolved – on top of the *active* theme when `theme` is `NULL`. An unknown slot name is an error, listing the valid ones. Each entry names only the fields to change;

unspecified fields are kept from the existing entry. Status slots take `glyph`, `width` and `color`; the close-line statuses (`done`, `warning`, `error`, `interrupted`) also take `text`. The `group` slot takes `bracket`, the `elapsed` slot governs the time column on close lines, and the two non-glyph slots `crumb` and `summary` carry `logtree_summary()`'s appearance. See the slot and field tables below for the complete set.

`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")`.

`glyph_gap` Number of spaces printed between a line's status glyph and its message text. `NULL` (the default) leaves the active setting alone; `1` is the built-in spacing, `0` butts the message straight against the glyph, and `2` or more airs the two columns apart. Applies to every line kind – step open, `Done` close, leaf, group header and the `with_logging()` run-summary line – so the message column stays aligned. Like `compact`, it applies to the active (console) theme and is cleared by a subsequent preset swap such as `logtree_theme("unicode")`.

`connector_gap` Number of spaces printed between a *leaf or close* line's own connector and its status glyph – `log_info()`/`log_warn()`/ etc. lines and a step's own `Done` line, never a step's *open* line (which always renders flush, at any `compact` density). `NULL` (the default) leaves the active setting alone, which tracks `col_gap` (so every built-in preset, and `compact = "medium"`/`"tight"`, render exactly as before). Set it explicitly to diverge from `col_gap` – e.g. pair it with `compact = "tight"` to keep every rail column flush (`col_gap = 0`, including step-open lines) while still spacing leaf/close glyphs off their own connector. Like `glyph_gap`, it applies to the active (console) theme and is cleared by a subsequent preset swap.

`wrap` Column budget a rendered line is wrapped to, instead of letting a long message run off the right edge. `TRUE` wraps at `cli::console_width()`, measured at render time so a terminal resized mid-run is picked up on its own – this is the console default; a positive number pins a fixed width; `FALSE` never wraps, letting long lines overflow. `NULL` (the argument default) leaves the active setting alone.

Continuation lines indent to the message column and carry the rails down, so a wrapped message still reads as one node of the tree: after a branch connector the vertical rail continues, after a corner it does not. A step-open line (and a group header) also rails its own glyph column, which is where its children hang – on a depth-1 root that is the only rail there is. A leaf's glyph column stays blank: nothing nests under a leaf. A token with no break opportunity – a long path, a URL – is split by display width rather than left to overflow, and a budget narrower than the tree is deep degrades to no wrapping rather than to an unusable one-column line. It applies to every line logtree renders, including the `logtree_summary()` digest and the `with_logging()` run-summary line.

Like `compact` and `glyph_gap`, it applies to the active (console) theme, and a subsequent preset swap returns it to the `TRUE` default. File sinks are never wrapped: they render through the ascii preset, and a file has no width to wrap to.

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`	`log_success()` leaf	`glyph`, `width`, `color`
`done`	a step's own close line on a clean close	`glyph`, `width`, `color`, `text`
`warning`	`log_warn()` / elevated step glyph	`glyph`, `width`, `color`, `text`
`error`	`log_error()` / elevated step glyph	`glyph`, `width`, `color`, `text`
`interrupted`	abnormal-exit (dimmed) glyph	`glyph`, `width`, `color`, `text`
`group`	group header marker	`glyph`, `color`, `bracket`
`elapsed`	the elapsed-time column printed on every close line	`show`, `min`, `color`, `slow`, `slow_color`
`trace`	the optional call-site column: where in your code a line came from	`show`, `capture`, `format`, `color`
`timestamp`	the optional wall-clock column printed in front of every line	`format`, `color`
`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`
`crumb`	`logtree_summary()` breadcrumb: the separator between path nodes	`glyph`, `color`, `path_color`
`summary`	`logtree_summary()` divider above the digest	`gap`, `rule`, `line`

`success` and `done` are separate slots that merely *look* the same by default (every preset ships the same tick in both): `success` styles the `log_success()` leaf line, `done` styles the `Done` line a step prints when it closes cleanly. Override one and the other is untouched. A step that closes elevated still renders `warning` / `error` / `interrupted`, so `done` only ever governs the clean close.

The same split governs the `text` field, the word a close line prints before its elapsed time. It is read from the closing status's own slot, falling back to `done`'s and then to the built-in `"Done"` – so `list(done = list(text = "Complete"))` renames every close line, while `list(error = list(text = "Failed"))` renames only the ones that went wrong. `success` has no `text` of its own precisely because a clean close reads `done`. `text` is a close-line concern only: it never touches the message a `log_warn()` or `log_error()` leaf prints.

The `trace` slot is off in every preset, and deliberately so: capturing a call site costs a frame walk and a source-reference lookup on every logged line, so you opt in and the default path pays nothing. Switch it on with `list(trace = list(show = "problems"))` to annotate only what went wrong, or `show = TRUE` for every line that can carry a call site. `show` also takes a vector of statuses, for when the bundle is too much: `show = "error"` annotates errors and leaves tolerated warnings bare, `show = c("error", "interrupted")` adds the steps that never finished. The column is appended to the message, so it wraps with it rather than shearing the tree.

The location – `{file}` and `{line}` together, separator included – is also emitted as one terminal hyperlink pointing at the file, at that line, so a click anywhere on it opens your editor there. Terminals without hyperlink support print the same text unlinked, and the escape has no printable width either way. `{file}` prints relative to the working directory where the source sits under it – a bare file name is not something a terminal can resolve.

On source references. {file} and {line} come from R's source references, which exist only when the code was parsed with keep.source = TRUE. That is the default in an interactive session and under devtools::load_all(), but **not** under plain Rscript and not for an installed package. {fn} is always available. Rather than print NA, the expander drops any whitespace-separated run of the template whose placeholders are all unavailable – so the default format degrades from pipeline.R:12 load_data() to load_data(), and a "{file}:{line}" format degrades to no column at all. Set options(keep.source = TRUE) at the top of a script if you want locations under Rscript.

Capturing and printing are separate: show decides what a line *prints*, capture = TRUE decides that call sites are *recorded* whatever show prints. list(trace = list(show = FALSE, capture = TRUE)) therefore leaves the tree exactly as it was while giving the digest and any "json" sink locations to work with. The order matters – capture happens as the run unfolds, so a call site not recorded then cannot be recovered afterwards.

The timestamp slot is off in every preset for a different reason: a tree read as it happens does not need to be told the time, and a column that is not there is one the message has room for. It is the log read *afterwards* that wants it – lining a run up against a monitoring graph, another service's log, or a report that something broke at about half past two. Switch it on with list(timestamp = list(format = "%H:%M:%S")).

The column is padded to a fixed width measured from a rendered sample, not from the format string, so a format whose width varies with the value ("%B", March one month and December the next) cannot shear the tree from one line to the next. It counts against the wrapping budget like any other column, and a wrapped message's continuation rows carry a *blank* column rather than a repeated time – one event happened once. logtree_summary()'s digest carries no timestamp at all: it replays events that already happened, so stamping those lines with the time the digest was printed would be a lie.

Two places outside the tree also report call sites when the slot is on: logtree_summary()'s digest lines, which apply the same status filter as the tree does, and the fn / file / line fields of a "json" sink (which carries them whenever they were captured, regardless of show). See logtree_sink_file() for pinning a file sink's column independently of the console's.

Fields (valid names inside a slot):

Field	Type	Accepted values
glyph	character(1)	Any string, including "". In package source, non-ASCII must
width	integer(1)	Rendered display width of glyph (1 for normal, 2 for emoji / v
color	character, NULL, or a named list on trace	One or more cli styles, or NULL for no styling. Named colors (
text	character(1)	Close-line status slots only (done, warning, error, interrup
show	logical(1), or character on trace	On elapsed: FALSE drops the elapsed-time column entirely, d
format	character(1)	On trace: a template for the call-site column over three place
capture	logical(1)	trace slot only. TRUE records a call site on every line even wh
min	numeric(1)	elapsed slot only. Hide times below this many seconds – min
slow	numeric(1) or NULL	elapsed slot only. Times at or over this many seconds count a
slow_color	character or NULL	elapsed slot only. Styles applied to a slow time in place of co
bracket	logical(1)	group slot only. TRUE wraps the header name in < >; default F
path_color	character or NULL	crumb slot only. Styles the breadcrumb's path nodes, setting th
gap	integer(1)	summary slot only. Blank lines printed above the digest; 0 prin
rule	logical(1) or character(1)	summary slot only. TRUE draws a cli::rule() labelled with th
line	integer(1) or character(1)	summary slot only. The rule's line, passed to cli::rule()'s l

Value

NULL, invisibly.

Examples

```
logtree_theme("ascii")
logtree_theme("unicode")

# No preset named: these merge onto the active theme, whichever it is.
logtree_theme(overrides = list(success = list(glyph = "*")))
# The close ("Done") tick is its own slot, restyled independently:
logtree_theme(list(done = list(glyph = "=", color = "silver")))
logtree_theme(list(group = list(glyph = "#", bracket = TRUE)))
logtree_theme(list(crumb = list(glyph = " / ", path_color = "cyan")))
logtree_theme(list(summary = list(gap = 2, rule = "Run report")))

# The word on a close line: every one at once, or only the failures.
logtree_theme(list(done = list(text = "Complete")))
logtree_theme(list(error = list(text = "Failed")))
logtree_theme(list(done = list(text = "{label} took {elapsed}")))
logtree_theme(list(done = list(text = "")))  # glyph + time only

# The elapsed-time column: hide the trivial, flag the slow.
logtree_theme(list(elapsed = list(min = 0.1)))
logtree_theme(list(elapsed = list(color = "silver", slow = 5,
                                  slow_color = "red")))
logtree_theme(list(elapsed = list(show = FALSE)))

# The call-site column: off by default, loudest on the lines that matter.
logtree_theme(list(trace = list(show = "problems")))
logtree_theme(list(trace = list(show = TRUE)))
logtree_theme(list(trace = list(show = "error")))          # errors only
# Record call sites but print none: the digest can still show them.
logtree_theme(list(trace = list(show = FALSE, capture = TRUE)))
logtree_theme(list(trace = list(show = c("error", "interrupted"))))
logtree_theme(list(trace = list(show = TRUE, format = "{fn}()")))
logtree_theme(list(trace = list(format = "{file}:{line}", color = "silver")))
logtree_theme(list(trace = list(show = FALSE)))  # back off again

# The wall-clock column: off by default, in front of every line when on.
logtree_theme(list(timestamp = list(format = "%H:%M:%S")))
logtree_theme(list(timestamp = list(format = "%Y-%m-%d %H:%M:%S",
                                    color = "silver")))
logtree_theme(list(timestamp = list(format = NULL)))   # back off again

# Naming one swaps the whole preset, clearing every override above.
logtree_theme("unicode")
logtree_theme("unicode", compact = "medium")
logtree_theme("unicode", compact = "tight")
```

```
# Tight rails, but with the glyph spaced off its connector.
logtree_theme("unicode", compact = "tight", connector_gap = 1)

# Spacing between the glyph and the message.
logtree_theme(glyph_gap = 0)   # tightest: no space after the glyph
logtree_theme(glyph_gap = 2)   # roomier message column

# Wrapping long messages (on by default, at the console width).
logtree_theme(wrap = 72)       # pin a fixed width
logtree_theme(wrap = FALSE)    # let long lines overflow instead
logtree_theme(wrap = TRUE)     # back to the console width

# The other two presets.
logtree_theme("minimal")       # no connectors: indentation only
logtree_theme("ci")            # [ok] / [warn] / [fail], no colour
logtree_theme("unicode")       # back to the default
```

`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.

Usage

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

Arguments

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

Details

This is the *default* for every sink; a sink registered with its own `threshold` ignores it (see `logtree_sink_file()` and `logtree_sink()`), which is what lets a debug-level log file coexist with an ordinary console.

Verbosity governs rendering only, never what the run remembers. A suppressed `log_warn()`/`log_error()` still elevates the enclosing step's close glyph, and still reaches the `logtree_summary()` digest – what is hidden is the leaf line's own text, not the fact that it happened.

Value

`NULL`, invisibly.

See Also

logtree_sink_file() and logtree_sink() for per-sink thresholds.

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, warnings = 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).

`warnings` Route R's own conditions into the tree as leaf lines? `FALSE` (the default) leaves `warning()` and `message()` exactly as they are. `TRUE` routes both; a subset such as `"warning"` or `"message"` routes only those.

Two consequences to weigh before switching it on:

- **Routed conditions are muffled.** A routed `warning()` becomes a leaf and stops there: it no longer reaches `warnings()`, the caller's own handlers, or stderr. That is the point – one record of the run rather than two halves – but it means the tree is now the only place that warning appears.
- **A routed warning elevates its step**, exactly as `log_warn()` does. So wrapping third-party code that warns freely will turn the enclosing steps yellow, which may be more honesty than you wanted.

In global mode the routing applies only while logtree steps are open, so a session-persistent handler cannot swallow warnings from unrelated code.

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
```

`warnings = TRUE` additionally routes R's *own* conditions into the tree: a `warning()` raised by wrapped code becomes a `log_warn()` leaf and a `message()` becomes a `log_info()` leaf, in place rather than on stderr, so the tree is a complete record of the run rather than half of one. It is opt-in because routing means **muffling** – see the argument's own documentation below for what that costs.

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")
})

# R's own conditions routed into the tree instead of onto stderr.
logtree_reset()
```

```
with_logging({
  log_step("Load data")
  warning("3 rows coerced to NA")
  message("using cached manifest")
}, warnings = TRUE)
```

Index