Package 'termr'

November 8, 2026

Title Modern Reactive Terminal Applications with Data Workflows

Version 1.0.2

Description Build modern reactive terminal applications for data workflows in R using reusable widgets, layouts, events, focus management and themes. A virtual screen buffer and incremental rendering update changed cells, while a headless driver supports deterministic application tests. Includes data tables that render visible rows on demand, lazy table sources, data inspection, SQL editing and background R workers with progress and streamed output.
Optional database support and static text, Markdown, HTML, SVG and JSON exports support analytical tools and reports without requiring an interactive terminal for static output.

License MIT + file LICENSE

URL https://github.com/menshikovs/termr

BugReports https://github.com/menshikovs/termr/issues

Encoding UTF-8

Depends R (>= 4.1.0)

Imports codetools, grDevices, R6, processx, ps, stats, utils

Suggests DBI, jsonlite, knitr, RSQLite, testthat (>= 3.0.0), withr

Config/testthat/edition 3

Config/roxygen2/version 8.1.1

NeedsCompilation no

Author Akims Mensikovs [aut, cre, cph]

Maintainer Akims Mensikovs <akimm7926@gmail.com>

Repository CRAN

Date/Publication 2026-11-08 13:40:02 UTC

Contents

animate . . . 4
Animation . . . 5
App . . . 6
app . . . 16
bind . . . 17
browse_data . . . 18
Button . . . 19
button . . . 21
cell . . . 22
Checkbox . . . 23
checkbox . . . 25
column . . . 26
command . . . 27
current_app . . . 28
DataProfile . . . 28
DataTable . . . 29
data_profile . . . 41
data_table . . . 42
db_connection . . . 44
db_explorer . . . 44
db_metadata . . . 45
db_query_source . . . 45
db_table_source . . . 46
diff_screen . . . 47
dropdown . . . 48
Event . . . 49
Grid . . . 56
grid_layout . . . 57
Horizontal . . . 58
Input . . . 59
input . . . 64
inspect_signal . . . 66
inspect_widget . . . 66
json_view . . . 67
KeyValue . . . 68
key_event . . . 69
key_value . . . 69
knit_termr . . . 70
Label . . . 71
label . . . 72
LogView . . . 73
log_view . . . 75
MarkdownView . . . 76
markdown_view . . . 77
Metric . . . 78
metric . . . 80

modal . . . 81
ModalScreen . . . 82
motion_reduced . . . 84
on . . . 84
OptionList . . . 85
option_list . . . 87
Panel . . . 87
panel . . . 88
patch_to_ansi . . . 89
ProcessView . . . 89
process_view . . . 91
ProgressBar . . . 92
progress_bar . . . 94
PropertyGrid . . . 95
property_grid . . . 96
RadioButton . . . 98
RadioSet . . . 99
radio_set . . . 101
reactive . . . 102
record_view . . . 102
region . . . 104
register_layout . . . 104
render_html . . . 105
render_markdown . . . 106
render_svg . . . 106
render_text . . . 107
render_widget . . . 108
Rule . . . 108
rule . . . 110
run . . . 110
run_example . . . 111
run_worker . . . 112
r_highlighter . . . 114
Screen . . . 115
ScreenBuffer . . . 115
screen_buffer . . . 119
screen_snapshot . . . 119
screen_snapshot_json . . . 120
ScrollView . . . 120
scroll_view . . . 124
Select . . . 126
set_timeout . . . 128
signals . . . 128
span . . . 130
Sparkline . . . 131
sparkline . . . 132
Spinner . . . 133
spinner . . . 135

SplitPane . . . 136
split_pane . . . 138
sql_editor . . . 139
sql_highlighter . . . 140
StatusBar . . . 140
status_bar . . . 142
strip_ansi . . . 143
style . . . 143
stylesheet . . . 145
table_filter . . . 146
table_source . . . 147
TabPane . . . 148
Tabs . . . 149
tabs . . . 152
terminal_capabilities . . . 153
termr_theme . . . 154
test_app . . . 155
TextArea . . . 156
text_area . . . 168
Timer . . . 169
Toast . . . 171
TreeNode . . . 172
TreeView . . . 175
tree_node . . . 178
tree_view . . . 179
unregister_layout . . . 180
Vertical . . . 181
vertical . . . 181
Widget . . . 182
widget . . . 193
widget_snapshot . . . 194
widget_snapshot_json . . . 194
Worker . . . 195
write_rendered . . . 196

Index **198**

`animate` *Animate a widget field*

Description

Changes a numeric field of a widget (e.g. `value` of a `progress_bar()`, `offset_y` of a `scroll_view()`, `count` of a custom widget) smoothly from its current value to `to`. Starting a new animation of the same field replaces the running one. Animations stop when the widget is removed or the app exits.

Usage

```
animate(
  widget,
  property,
  to,
  duration = 0.3,
  easing = "out_cubic",
  from = NULL,
  on_complete = NULL
)
```

Arguments

`widget`	A widget attached to a running app.
`property`	Name of a numeric field.
`to`	Target value.
`duration`	Seconds.
`easing`	`"linear"`, `"in_quad"`, `"out_quad"`, `"in_out_quad"`, `"in_cubic"`, `"out_cubic"` (default) or `"in_out_cubic"`.
`from`	Start value (default: the current value).
`on_complete`	Optional `function(widget, app)`.

Value

An `Animation` handle with `cancel()`.

Examples

```
bar <- progress_bar(id = "progress")
pilot <- test_app(app(bar), width = 30, height = 1)
animate(pilot$query_one("#progress"), "value", to = 1, duration = 0.5)
pilot$advance(1)
bar$value
pilot$stop()
```

`Animation` *Animation handle*

Description

Returned by `animate()`.

Public fields

`widget`, `property` What is animated.

`from`, `to` Start and end values.

`duration` Seconds.

`easing` Easing name.

`start` Clock time of the start.

`active` Still running?

`on_complete` `function(widget, app)` or `NULL`.

Methods

Public methods:

- `Animation$new()`
- `Animation$cancel()`
- `Animation$clone()`

`Animation$new()`: Create an animation (use `animate()`).

Usage:

```
Animation$new(widget, property, from, to, duration, easing, start, on_complete)
```

Arguments:

`widget`, `property`, `from`, `to`, `duration`, `easing`, `start`, `on_complete` See fields.

`Animation$cancel()`: Stop the animation where it is.

Usage:

```
Animation$cancel()
```

`Animation$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

App *Application*

Description

An App owns the widget tree (through its Screen), the event queue, timers, key bindings, focus and the terminal driver. Create one with `app()` and start it with `run()`.

Event loop

Each tick of the loop waits for input (or the next timer), turns input into events, fires due timers, dispatches every queued event and then, if any widget was invalidated, repaints the screen once: the new frame is diffed against the previous one and only changed cells are written.

Event dispatch

An event goes to its target widget (key events: the focused widget) and bubbles up through the ancestors to the app. At each widget the `on_<type>` method runs, then handlers added with `widget$on()`, then - for key events - the widget's key bindings. At the app, handlers added with `app$on()` / `on()` run, then app bindings. `event$stop()` stops propagation.

Public fields

`title` Optional application title.

`reduce_motion` `NULL` (follow the option / environment variable), `TRUE` or `FALSE`: reduce decorative animation? See `motion_reduced()`.

`clipboard` Text copied by widgets (Ctrl+C / Ctrl+X in inputs). This is the app's own clipboard, not the operating system's.

Active bindings

`screen` The current Screen.

`screens` The screen stack, bottom first.

`focused` The focused widget, or `NULL`.

`theme` The `termr_theme()` (assign a theme or a theme name).

`stylesheets` The stylesheets of the app.

`hovered` The widget under the mouse pointer, or `NULL`.

`size` Terminal size: `c(width =, height =)`.

`running` Is the app running?

`last_paint` Instrumentation of the last repaint: screen cells, repainted cells, dirty rectangles, whether it was a full repaint and the ANSI bytes written. For benchmarks and debugging.

`profile_last_frame` Optional timings and structural counters; enable with `options(termr.profile = TRUE)`. Times are milliseconds.

`frame_stats` Number of full and incremental repaints and of layout passes so far.

`frame` The last rendered ScreenBuffer.

Methods

Public methods:

- `App$new()`
- `App$run()`
- `App$exit()`
- `App$query()`

- App$query_one()
- App$mount()
- App$on()
- App$bind()
- App$log_events()
- App$event_log()
- App$toggle_debug()
- App$add_command()
- App$show_help()
- App$bindings()
- App$post()
- App$post_message()
- App$clipboard_write()
- App$capture_mouse()
- App$release_mouse()
- App$run_action()
- App$request_repaint()
- App$refresh()
- App$set_focus()
- App$add_stylesheet()
- App$clear_stylesheets()
- App$stylesheet_style()
- App$push_screen()
- App$pop_screen()
- App$switch_screen()
- App$run_worker()
- App$run_process()
- App$workers()
- App$notify()
- App$notifications()
- App$scroll_into_view()
- App$focus_next()
- App$focus_previous()
- App$focus_chain()
- App$set_timeout()
- App$set_interval()
- App$call_later()
- App$print()
- App$clone()

`App$new()`: Create an app. See `app()`.

Usage:

```
App$new(
  ...,
  bindings = list(),
  actions = list(),
  title = NULL,
  mouse = TRUE,
  stylesheet = NULL,
  theme = "default",
  debug = FALSE,
  reduce_motion = NULL
)
```

Arguments:

`...` Widgets, `on()` handlers and `bind()` bindings.

`bindings` A list of `bind()` bindings.

`actions` A named list of functions `function(app)`.

`title` Optional title.

`mouse` Enable mouse input?

`stylesheet` A `stylesheet()`, stylesheet text or a file path (or a list of them).

`theme` A `termr_theme()` or the name of a built-in theme.

`debug` Bind F12 to the debug overlay?

`reduce_motion` `TRUE` to turn decorative animation off, `NULL` to follow `options(termr.reduce_motion)` / the `TERMR_REDUCE_MOTION` environment variable.

`App$run()`: Run the app until it exits. Restores the terminal on exit, on error and on interrupt.

Usage:

```
App$run(driver = NULL)
```

Arguments:

`driver` A terminal driver; detected automatically by default.

Returns: The value passed to `exit()`, invisibly.

`App$exit()`: Ask the app to stop after the current event.

Usage:

```
App$exit(value = NULL)
```

Arguments:

`value` Value returned by `run()`.

`App$query()`: Find widgets matching a selector.

Usage:

```
App$query(selector)
```

Arguments:

`selector` A selector such as `"Button"`, `"#id"`, `".class"`.

`App$query_one()`: The first widget matching a selector (error if none).

Usage:

```
App$query_one(selector)
```

Arguments:

`selector` A selector.

`App$mount()`: Mount widgets on the screen.

Usage:

```
App$mount(...)
```

Arguments:

`...` Widgets.

`App$on()`: Add an event handler, see `on()`.

Usage:

```
App$on(type, selector = NULL, handler)
```

Arguments:

`type` Event type.

`selector` Optional selector matched against the sender.

`handler` `function(event, app)`.

`App$bind()`: Add an app-level key binding, see `bind()`.

Usage:

```
App$bind(key, action = NULL, description = NULL)
```

Arguments:

`key` Key name or a binding created by `bind()`.

`action` Action name or function.

`description` Optional description.

`App$log_events()`: Record dispatched events (for debugging). Off by default, so normal apps pay nothing.

Usage:

```
App$log_events(enable = TRUE, size = 200L)
```

Arguments:

`enable` Start (`TRUE`) or stop (`FALSE`) recording.

`size` Number of events kept.

`App$event_log()`: The recorded events as a data frame (see `log_events()`).

Usage:

```
App$event_log()
```

`App$toggle_debug()`: Show or hide the debug overlay (also F12 with `app(debug = TRUE)`).

Usage:

```
App$toggle_debug()
```

`App$add_command()`: Add a command to the command palette (Ctrl+P) and the help screen (F1). See `command()`.

Usage:

```
App$add_command(
  label,
  action = NULL,
  category = NULL,
  shortcut = NULL,
  enabled = NULL,
  description = NULL
)
```

Arguments:

`label` Text shown in the palette, or a `command()` object.

`action` An action name or `function(app)`.

`category`, `shortcut`, `enabled`, `description` See `command()`.

`App$show_help()`: Show the help screen (F1): key bindings of the focused widget, its ancestors and the app, and the app's commands.

Usage:

```
App$show_help()
```

`App$bindings()`: All app-level bindings.

Usage:

```
App$bindings()
```

`App$post()`: Queue an event for dispatch.

Usage:

```
App$post(event)
```

Arguments:

`event` An Event.

`App$post_message()`: Send a custom message from the app (bubbles from the screen).

Usage:

```
App$post_message(type, data = list())
```

Arguments:

`type` Message name.

`data` List of values.

`App$clipboard_write()`: Copy text to the clipboard. The app clipboard (`app$clipboard`) is always set; when `system = TRUE` and the terminal supports OSC 52 the text is also sent to the system clipboard (write-only: termr never reads the system clipboard). Long text is truncated to 100 kB for the terminal.

Usage:

```
App$clipboard_write(text, system = TRUE)
```

Arguments:

`text` A string.

`system` Also write to the system clipboard?

Returns: `TRUE` (invisibly) if the text was sent to the terminal.

`App$capture_mouse()`**:** Send all mouse events to `widget` (even outside its region) until `release_mouse()` or the next button release. Used for drags that start elsewhere, e.g. splitters.

Usage:

```
App$capture_mouse(widget)
```

Arguments:

`widget` A mounted widget.

`App$release_mouse()`**:** Stop capturing the mouse.

Usage:

```
App$release_mouse()
```

`App$run_action()`**:** Run an action by name (or a function).

Usage:

```
App$run_action(action, widget = NULL)
```

Arguments:

`action` Action name such as `"quit"`, or a function.

`widget` Widget where the lookup starts (`NULL`: the app).

`App$request_repaint()`**:** Schedule a repaint at the end of the current tick.

Usage:

```
App$request_repaint(widget = NULL, full = is.null(widget), layout = TRUE)
```

Arguments:

`widget` The widget that changed, if known. Without it the next repaint redraws everything; with it only the rows of changed widgets are repainted when the layout did not change.

`full` Force a full repaint?

`layout` Does the change possibly affect sizes or positions? When `FALSE` and nothing else changed the layout pass is skipped.

`App$refresh()`**:** Redraw the whole screen (e.g. after external output).

Usage:

```
App$refresh()
```

`App$set_focus()`**:** Give focus to a widget (or remove focus with `NULL`).

Usage:

```
App$set_focus(widget)
```

Arguments:

`widget` A focusable widget in this app, or `NULL`.

Returns: `TRUE` if focus changed.

`App$add_stylesheet()`: Add a stylesheet; its rules apply to all screens.

Usage:

```
App$add_stylesheet(sheet)
```

Arguments:

`sheet` A `stylesheet()`, stylesheet text or a file path.

`App$clear_stylesheets()`: Remove all stylesheets.

Usage:

```
App$clear_stylesheets()
```

`App$stylesheet_style()`: The merged stylesheet rules that apply to a widget.

Usage:

```
App$stylesheet_style(widget, states = widget$pseudo_states())
```

Arguments:

`widget` A widget.

`states` Its pseudo states.

`App$push_screen()`: Show a screen on top of the current one. A widget is wrapped in a new Screen. Use `modal()` for dialogs.

Usage:

```
App$push_screen(screen, callback = NULL)
```

Arguments:

`screen` A Screen (or ModalScreen) or a widget.

`callback` Optional `function(result, app)` called when the screen is dismissed / popped with a result.

Returns: The screen, invisibly.

`App$pop_screen()`: Remove the top screen (or a given screen) and return to the one below, restoring its focus.

Usage:

```
App$pop_screen(result = NULL, screen = self$screen)
```

Arguments:

`result` Passed to the screen's callbacks.

`screen` The screen to remove (default: the top screen).

`App$switch_screen()`: Replace the top screen.

Usage:

```
App$switch_screen(screen)
```

Arguments:

`screen` A Screen or a widget.

`App$run_worker()`: Run a function in a background R process. See `run_worker()` for the arguments.

Usage:

```
App$run_worker(
  fn,
  args = list(),
  on_complete = NULL,
  on_error = NULL,
  on_progress = NULL,
  name = NULL,
  owner = NULL,
  packages = character(),
  inline = FALSE,
  on_stdout = NULL,
  on_stderr = NULL,
  timeout = NULL
)
```

Arguments:

`fn`, `args`, `on_complete`, `on_error`, `on_progress`, `name`, `owner`, `packages`, `inline` See `run_worker()`.

`on_stdout`, `on_stderr`, `timeout` See `run_worker()`.

`App$run_process()`: Run an external program in the background, without a shell. See `run_process()`.

Usage:

```
App$run_process(
  command,
  args = character(),
  on_complete = NULL,
  on_error = NULL,
  on_stdout = NULL,
  on_stderr = NULL,
  on_exit = NULL,
  timeout = NULL,
  wd = NULL,
  env = NULL,
  name = NULL,
  owner = NULL
)
```

Arguments:

`command`, `args`, `on_complete`, `on_error`, `on_stdout`, `on_stderr`, `on_exit`, `timeout`, `wd`, `env`, `name`, `owner` See `run_process()`.

`App$workers()`: Running workers.

Usage:

```
App$workers()
```

`App$notify()`: Show a notification (toast) in the bottom-right corner.

Usage:

```
App$notify(message, title = NULL, severity = "information", timeout = 3)
```

Arguments:

`message` Text.

`title` Optional title.

`severity` `"information"`, `"success"`, `"warning"` or `"error"`.

`timeout` Seconds before it disappears (`Inf`: until clicked).

Returns: The `Toast` widget, invisibly.

`App$notifications()`: Active notifications (toasts).

Usage:

```
App$notifications()
```

`App$scroll_into_view()`: Scroll every scroll view containing `widget` so that it becomes visible (done at the next repaint).

Usage:

```
App$scroll_into_view(widget)
```

Arguments:

`widget` A widget of this app.

`App$focus_next()`: Move focus to the next focusable widget.

Usage:

```
App$focus_next()
```

`App$focus_previous()`: Move focus to the previous focusable widget.

Usage:

```
App$focus_previous()
```

`App$focus_chain()`: Focusable widgets in focus order.

Usage:

```
App$focus_chain()
```

`App$set_timeout()`: Call `callback(app)` once after `delay` seconds.

Usage:

```
App$set_timeout(delay, callback)
```

Arguments:

`delay` Seconds.

`callback` Function.

`App$set_interval()`: Call `callback(app)` every `interval` seconds.

Usage:

```
App$set_interval(interval, callback)
```

Arguments:

`interval` Seconds.

`callback` Function.

`App$call_later()`: Call `callback(app)` at the start of the next tick.

Usage:

```
App$call_later(callback)
```

Arguments:

`callback` Function.

`App$print()`: Print a summary of the app.

Usage:

```
App$print(...)
```

Arguments:

`...` Ignored.

`App$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

---

app *Create an application*

Description

Create an application

Usage

```
app(
  ...,
  bindings = list(),
  actions = list(),
  title = NULL,
  mouse = TRUE,
  stylesheet = NULL,
  theme = "default",
  debug = FALSE,
  reduce_motion = NULL
)
```

Arguments

`...`	Widgets to show, plus optional `on()` event handlers and `bind()` key bindings.
`bindings`	A list of `bind()` key bindings.
`actions`	A named list of action functions `function(app)` that bindings can refer to by name.
`title`	Optional title.
`mouse`	Enable mouse input (clicks, wheel, hover)?
`stylesheet`	A `stylesheet()`, stylesheet text, a file path, or a list of them.
`theme`	A `termr_theme()` or a built-in theme name (`"default"`, `"dark"`, `"light"`).
`debug`	Bind F12 to a debug overlay (focused and hovered widget, last event, repaint statistics)?
`reduce_motion`	Turn decorative animation off? `NULL` follows `motion_reduced()`.

Value

An App. Start it with `run()`.

Built-in bindings: `tab` / `shift+tab` move focus, `ctrl+c` quits.

Examples

```
hello <- app(
  vertical(
    label("What is your name?"),
    input(id = "name"),
    button("Say hello", id = "hello"),
    label("", id = "output")
  ),
  on("button.pressed", "#hello", function(event, app) {
    name <- app$query_one("#name")$value
    app$query_one("#output")$update(paste("Hello", name))
  }),
  bind("escape", "quit")
)
if (interactive()) run(hello)
```

`bind` *Key bindings*

Description

`bind()` maps a key (or several keys separated by commas) to an action. Bindings can be attached to the app (`app()`), to the screen, to a widget type (`widget()`) or to a single widget (`widget$bind()`). When a key is pressed, bindings are searched from the focused widget up to the app, so the most local binding wins.

Usage

```
bind(key, action, description = NULL)
```

Arguments

key	Key name such as "q", "ctrl+c", "shift+tab", "f1", or several keys: "q,escape".
action	Action name or a function. Functions bound on the app are called as action(app); functions bound on a widget as action(widget, app).
description	Optional human readable description (for help screens and footers).

Details

An action is either the name of an action or a function. Named actions are looked up on the widget that owns the binding, then on its ancestors, then on the app. Prefix a name with "app." to call an app action directly. Built-in app actions are "quit", "focus_next", "focus_previous" and "refresh".

Value

An object of class termr_binding.

Examples

```
bind("q", "quit")
bind("ctrl+r", function(app) app$refresh(), "Redraw")
```

---

browse_data *Browse a data frame in the terminal*

Description

browse_data() opens an interactive viewer for a data frame: summary metrics, a virtualised table (fast for millions of rows), information about the selected column, search across all columns, sorting, and row details. data_browser() builds the same app without running it, e.g. to customise or test it.

Usage

```
browse_data(data, title = deparse(substitute(data)), ...)

data_browser(data, title = deparse(substitute(data)), ...)
```

Arguments

data	A data frame or matrix.
title	Title shown at the top (defaults to the expression).
...	Passed to data_table() (e.g. formatters, columns).

Details

Keys: arrows / Page Up / Page Down move the cell cursor; `/` focuses the search box (Enter applies it, Escape in the box clears it); `s` sorts by the current column (again: descending); `r` resets sorting and search; Enter shows the row; `q` quits.

Value

`browse_data()` returns `NULL` invisibly; `data_browser()` returns an App.

Examples

```
viewer <- data_browser(mtcars)
pilot <- test_app(viewer, 100, 30)
pilot$press("s")
pilot$stop()
if (interactive()) browse_data(mtcars)
```

`Button` *Button widget*

Description

A focusable button. See `button()`.

Super class

`Widget` -> `Button`

Public fields

`paint_states` Pressing only repaints.

`focusable` Buttons can be focused.

`variant` Colour variant.

`press_duration` Seconds the pressed state stays visible.

Active bindings

`label` The button text (reactive).

`pressed` Is the button showing its pressed state?

Methods

Public methods:

- Button$new()
- Button$default_style()
- Button$default_bindings()
- Button$render()
- Button$press()
- Button$on_click()
- Button$action_press()
- Button$pseudo_states()
- Button$clone()

`Button$new()`: Create a button.

Usage:

```
Button$new(
  label = "Button",
  id = NULL,
  classes = NULL,
  style = NULL,
  disabled = FALSE,
  variant = "default"
)
```

Arguments:

`label` Button text, or a function of no arguments that returns it (re-evaluated when the `signal()`s it reads change).

`id`, `classes`, `style`, `disabled` See Widget.

`variant` One of `"default"`, `"primary"`, `"success"`, `"warning"`, `"error"`.

`Button$default_style()`: The built-in style.

Usage:

```
Button$default_style()
```

`Button$default_bindings()`: Enter and space press the button.

Usage:

```
Button$default_bindings()
```

`Button$render()`: The button text.

Usage:

```
Button$render()
```

`Button$press()`: Press the button: shows the pressed state briefly and sends a `"button.pressed"` message. Does nothing when disabled.

Usage:

```
Button$press()
```

Returns: TRUE if the button was pressed.

`Button$on_click()`: A left click presses the button.

Usage:

```
Button$on_click(event)
```

Arguments:

`event` A `MouseEvent`.

`Button$action_press()`: Action used by the default bindings.

Usage:

```
Button$action_press()
```

`Button$pseudo_states()`: Active states: focus, disabled and pressed.

Usage:

```
Button$pseudo_states()
```

`Button$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`button` *Button*

Description

A focusable button. Pressing Enter or Space while it has focus sends a `"button.pressed"` message, which you can handle with `on("button.pressed", "#id", function(event, app) ...)`.

Usage

```
button(
  label = "Button",
  id = NULL,
  classes = NULL,
  style = NULL,
  disabled = FALSE,
  variant = "default",
  on_press = NULL
)
```

Arguments

label	Button text.
id	Optional identifier (for #id selectors).
classes	Optional classes (for .class selectors).
style	A style(). State styles focus, disabled and states = list(pressed = ...) are supported.
disabled	Disabled buttons cannot be focused or pressed.
variant	Colour variant: "default", "primary", "success", "warning" or "error".
on_press	Optional function() (or function(event, app)) called when the button is pressed.

Value

A Button widget.

Examples

```
button("Run", id = "run", variant = "primary")
```

---

cell *A single terminal cell*

Description

Cells are the unit of the virtual screen. Each cell holds one character (a wide character such as a CJK ideograph occupies two cells; the second one is a continuation cell whose char is ""), colours and text attributes.

Usage

```
cell(
  char = " ",
  fg = NULL,
  bg = NULL,
  bold = FALSE,
  italic = FALSE,
  underline = FALSE,
  reverse = FALSE,
  dim = FALSE,
  strike = FALSE
)
```

Arguments

`char`	A single character.
`fg`, `bg`	Foreground and background colours. `NULL` means the terminal default. See `style()` for accepted colour formats.
`bold`, `italic`, `underline`, `reverse`, `dim`, `strike`	Text attributes.

Details

The framebuffer stores cells column-wise in matrices for speed; `cell()` is the value-level representation used for inspection and tests.

Value

An object of class `termr_cell`.

Examples

```
cell("A", fg = "red", bold = TRUE)
```

`Checkbox` *Checkbox widget*

Description

A labelled boolean toggle. See `checkbox()`.

Super class

`Widget` -> `Checkbox`

Public fields

`focusable` Checkboxes can be focused.

Active bindings

`value` `TRUE` or `FALSE` (reactive; sends `"checkbox.changed"`).

`label` The label (reactive).

Methods

Public methods:

- Checkbox$new()
- Checkbox$default_style()
- Checkbox$default_bindings()
- Checkbox$toggle()
- Checkbox$action_toggle()
- Checkbox$on_click()
- Checkbox$render()
- Checkbox$clone()

`Checkbox$new()`: Create a checkbox. See `checkbox()`.

Usage:

```
Checkbox$new(
  label = "",
  value = FALSE,
  id = NULL,
  classes = NULL,
  style = NULL,
  disabled = FALSE
)
```

Arguments:

`label` Label.

`value` Initial state.

`id`, `classes`, `style`, `disabled` See Widget.

`Checkbox$default_style()`: The built-in style.

Usage:

```
Checkbox$default_style()
```

`Checkbox$default_bindings()`: Space and Enter toggle.

Usage:

```
Checkbox$default_bindings()
```

`Checkbox$toggle()`: Toggle the value (sends `"checkbox.changed"`).

Usage:

```
Checkbox$toggle()
```

`Checkbox$action_toggle()`: Action used by the bindings.

Usage:

```
Checkbox$action_toggle()
```

`Checkbox$on_click()`: Clicking toggles.

Usage:

`Checkbox$on_click(event)`

Arguments:

`event` A `MouseEvent`.

`Checkbox$render()`: The box and the label.

Usage:

```
Checkbox$render()
```

`Checkbox$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`checkbox` *Checkbox*

Description

A labelled toggle. Space, Enter or a click toggles it and sends `"checkbox.changed"` (`event$data$value`).

Usage

```
checkbox(
  label = "",
  value = FALSE,
  id = NULL,
  classes = NULL,
  style = NULL,
  disabled = FALSE
)
```

Arguments

`label`	Label text.
`value`	Initial state.
`id`	Optional identifier.
`classes`	Optional classes.
`style`	A `style()`.
`disabled`	Disabled checkboxes cannot be toggled.

Value

A `Checkbox` widget with a reactive `value`.

Examples

```
checkbox("Enable logs", value = TRUE, id = "logs")
```

column *Column options for data tables*

Description

Column options for data tables

Usage

```
column(
  label = NULL,
  width = NULL,
  align = NULL,
  formatter = NULL,
  min_width = NULL,
  max_width = NULL,
  visible = TRUE,
  sortable = TRUE,
  editable = NULL
)
```

Arguments

label	Header text (defaults to the column name).
width	Fixed width in cells; NULL estimates it from the data.
align	"left", "right" or "center"; numbers default to right.
formatter	function(x) turning a vector of values into strings.
min_width, max_width	Limits for the estimated width.
visible	Show the column? (Hidden columns stay in the data; see set_column_visible().)
sortable	Can the table be sorted by this column?
editable	Can cells be edited (when the table is editable)? NULL: yes for numeric, logical, character, factor and date columns.

Value

An object of class termr_column.

Examples

```
column(label = "Score", width = 8, align = "right",
       formatter = function(x) sprintf("%.1f", x))
```

command *Commands*

Description

A command is something the user can do: it has a label, an action, an optional category and keyboard shortcut, and may be enabled or disabled depending on the app state. The command palette (Ctrl+P) and the help screen (F1) are built from the same list: commands added with `app$add_command()`, the described key bindings of the focused widget, its ancestors and the app, and the app's named actions. Existing `bind()` bindings keep working and appear in both.

Usage

```
command(
  label,
  action,
  category = NULL,
  shortcut = NULL,
  enabled = NULL,
  description = NULL
)
```

Arguments

`label`	Text shown in the palette and the help screen.
`action`	An action name or a `function(app)`.
`category`	Optional group name, e.g. `"File"`.
`shortcut`	Optional key shown next to the command (documentation only; use `bind()` to create the binding).
`enabled`	`NULL` (always) or a `function(app)` returning `TRUE` / `FALSE`. Disabled commands are dimmed in the palette and cannot run.
`description`	Optional longer text for the help screen.

Value

An object of class `termr_command`.

Examples

```
command("Save", action = "save", category = "File", shortcut = "ctrl+s")
```

current_app *The running app*

Description

The running app

Usage

```
current_app()
```

Value

The App that is currently running, or NULL.

DataProfile *DataProfile widget*

Description

See data_profile().

Super classes

Widget -> Vertical -> DataProfile

Public fields

sample The deterministic sample size, or NULL for full scan.

top_n Number of frequent text values shown.

Active bindings

data The vector being summarized (reactive).

summary Computed summary statistics and histogram data.

Methods

Public methods:

- DataProfile$new()
- DataProfile$default_style()
- DataProfile$clone()

DataProfile$new(): Create a vector profile. See data_profile().

Usage:

```
DataProfile$new(
  data,
  name = "value",
  sample = NULL,
  top_n = 5L,
  id = NULL,
  classes = NULL,
  style = NULL,
  disabled = FALSE
)
```

Arguments:

`data` An in-memory supported vector.

`name` Heading.

`sample` Optional deterministic sample size.

`top_n` Frequent values to show.

`id`, `classes`, `style` See Widget.

`disabled` Is the widget disabled?

`DataProfile$default_style()`: The built-in style.

Usage:

```
DataProfile$default_style()
```

`DataProfile$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`DataTable` *DataTable widget*

Description

A virtualised table for data frames or lazy table sources. See `data_table()`.

Super class

`Widget` -> `DataTable`

Public fields

`paint_states` Cursor and scroll changes only repaint.

`focusable` Tables can be focused.

`cursor_type` `"row"`, `"cell"` or `"none"`.

`show_header` Show the header row?

`zebra` Stripe alternate rows?

`row_style` `NULL` or `function(row, data)` returning a style for a data row (`row` is the original row number or source position).

`cell_style` `NULL` or `function(value, row, column)` returning a style for a cell.

`frozen_columns` Number of leading columns that stay visible while the others scroll horizontally.

`header_sort` Clicking a header sorts by that column (click again: descending, then unsorted; Shift+click adds a column)?

`editable` Allow editing cells (Enter / double click)?

`on_edit` `NULL` or `function(row, column, value)` called before a cell edit is applied; return `NULL` / `TRUE` to accept or a message to reject the edit.

Active bindings

`data` The data frame or source (assigning calls `set_data()`).

`row_count` Number of rows in the current view.

`column_names` Names of the data columns.

`cursor_row` Cursor position in the view (0 = none).

`cursor_column` Cursor column number (cell cursor; 0 = none).

`offset_row`, `offset_column` Scroll offsets (rows / columns).

`view` Row numbers or source view positions in display order.

`sort_state` `NULL` or a data frame (`column`, `decreasing`) of the current sort keys.

`match_hit` `NULL` or the current search result (`position`, `row`, `column`).

Methods

Public methods:

- `DataTable$new()`
- `DataTable$default_style()`
- `DataTable$default_bindings()`
- `DataTable$set_data()`
- `DataTable$refresh()`
- `DataTable$close()`
- `DataTable$source_stats()`
- `DataTable$sort()`
- `DataTable$clear_sort()`
- `DataTable$toggle_sort()`

- DataTable$filter()
- DataTable$filter_columns()
- DataTable$move_cursor()
- DataTable$scroll_to_row()
- DataTable$selected_row()
- DataTable$row_data()
- DataTable$selected_data()
- DataTable$visible_data()
- DataTable$reorder_column()
- DataTable$select_range()
- DataTable$copy_selection()
- DataTable$goto_row()
- DataTable$column_width()
- DataTable$set_column_width()
- DataTable$auto_size_column()
- DataTable$auto_size_all()
- DataTable$set_column_visible()
- DataTable$column_info()
- DataTable$find()
- DataTable$find_next()
- DataTable$find_previous()
- DataTable$clear_find()
- DataTable$edit_cell()
- DataTable$set_cell()
- DataTable$action_cursor_up()
- DataTable$action_cursor_down()
- DataTable$action_extend_selection_up()
- DataTable$action_extend_selection_down()
- DataTable$action_move_column_left()
- DataTable$action_move_column_right()
- DataTable$action_copy_selection()
- DataTable$action_cursor_left()
- DataTable$action_cursor_right()
- DataTable$action_page_up()
- DataTable$action_page_down()
- DataTable$action_first_row()
- DataTable$action_last_row()
- DataTable$action_open_find()
- DataTable$action_open_goto()
- DataTable$action_find_next()
- DataTable$action_find_previous()
- DataTable$action_widen_column()

- DataTable$action_narrow_column()
- DataTable$action_sort_column()
- DataTable$action_activate()
- DataTable$on_mouse_down()
- DataTable$on_drag_move()
- DataTable$on_drag_end()
- DataTable$on_key()
- DataTable$on_paste()
- DataTable$on_mouse_scroll()
- DataTable$content_width()
- DataTable$content_height()
- DataTable$paint()
- DataTable$clone()

`DataTable$new()`: Create a table. See `data_table()`.

Usage:

```
DataTable$new(
  data,
  cursor = "row",
  columns = NULL,
  formatters = NULL,
  row_names = NA,
  header = TRUE,
  zebra = FALSE,
  row_style = NULL,
  cell_style = NULL,
  max_column_width = 40L,
  frozen_columns = 0L,
  header_sort = FALSE,
  editable = FALSE,
  on_edit = NULL,
  id = NULL,
  classes = NULL,
  style = NULL,
  disabled = FALSE
)
```

Arguments:

`data` A data frame, matrix, or `table_source()`.

`cursor` Cursor type.

`columns` Named list of `column()` options.

`formatters` Named list of formatter functions.

`row_names` Show row names? `NA`: only when they are not 1..n.

`header` Show the header?

`zebra` Stripe rows?

`row_style, cell_style` Style hooks.

`max_column_width` Upper limit for estimated widths.

`frozen_columns`, `header_sort`, `editable`, `on_edit` See the fields.

`id`, `classes`, `style`, `disabled` See Widget.

`filters` Named column filters for `filter_columns()`.

`DataTable$default_style()`: The built-in style.

Usage:

```
DataTable$default_style()
```

`DataTable$default_bindings()`: Navigation keys and Enter.

Usage:

```
DataTable$default_bindings()
```

`DataTable$set_data()`: Replace the data. Resets the view, cursor and scroll.

Usage:

```
DataTable$set_data(data)
```

Arguments:

`data` A data frame, matrix, or `table_source()`.

`DataTable$refresh()`: Re-read the source row count and clear cached rows.

Usage:

```
DataTable$refresh()
```

`DataTable$close()`: Explicitly close this table's source resource.

Usage:

```
DataTable$close()
```

`DataTable$source_stats()`: Read structural fetch/cache counters for diagnostics.

Usage:

```
DataTable$source_stats()
```

`DataTable$sort()`: Sort the view by one or more columns (stable; `NA`s last). Sorting always starts from the unsorted view (after `filter()`), so `sort(c("cyl", "mpg"))` sorts by cyl, then mpg.

Usage:

```
DataTable$sort(by, decreasing = FALSE)
```

Arguments:

`by` Column names or numbers.

`decreasing` Sort in decreasing order? One value, or one per column.

`DataTable$clear_sort()`: Remove the sort order (back to the order before sorting).

Usage:

```
DataTable$clear_sort()
```

`DataTable$toggle_sort()`: Header-click behaviour: sort by a column ascending, then descending, then unsorted.

Usage:

```
DataTable$toggle_sort(column, add = FALSE)
```

Arguments:

`column` Column name or number.

`add` Add the column to the existing sort (secondary key)?

`DataTable$filter()`: Show a subset of rows (in the given order), or all rows.

Usage:

```
DataTable$filter(rows = NULL)
```

Arguments:

`rows` Row numbers of the data, a logical vector over the rows, or `NULL` for all rows.

`DataTable$filter_columns()`: Filter named columns; all predicates are combined with AND and evaluated in row chunks. A predicate may be a `table_filter()` or `function(values) logical`.

Usage:

```
DataTable$filter_columns(filters = NULL)
```

Arguments:

`filters` Named list by column; `NULL` clears column filters.

`DataTable$move_cursor()`: Move the cursor (and scroll so it is visible).

Usage:

```
DataTable$move_cursor(row = NULL, column = NULL)
```

Arguments:

`row` Position in the view.

`column` Column name or number (cell cursor).

`DataTable$scroll_to_row()`: Scroll so that a view position is the first visible row.

Usage:

```
DataTable$scroll_to_row(row)
```

Arguments:

`row` Position in the view.

`DataTable$selected_row()`: The data-frame row number or source row key under the cursor (or `NA`).

Usage:

```
DataTable$selected_row()
```

`DataTable$row_data()`: One data row as a named list.

Usage:

```
DataTable$row_data(row)
```

Arguments:

`row` Original row number.

`DataTable$selected_data()`: Data for selected view rows, in view order. A single cursor row is returned when no range is selected.

Usage:

```
DataTable$selected_data()
```

`DataTable$visible_data()`: Data in the current viewport, in display order.

Usage:

```
DataTable$visible_data()
```

`DataTable$reorder_column()`: Move a column in the display order.

Usage:

```
DataTable$reorder_column(column, to)
```

Arguments:

`column` Column name or number.

`to` New 1-based position in the complete column order.

`DataTable$select_range()`: Select a range of rows by view position.

Usage:

```
DataTable$select_range(from, to = from)
```

Arguments:

`from, to` 1-based positions in the view.

`DataTable$copy_selection()`: Copy selected rows as tab-separated values.

Usage:

```
DataTable$copy_selection(headers = TRUE, system = TRUE)
```

Arguments:

`headers` Include column names?

`system` Also write to the system clipboard when supported?

`DataTable$goto_row()`: Jump to a row of the view (and scroll to it).

Usage:

```
DataTable$goto_row(row)
```

Arguments:

`row` Position in the view (1-based).

`DataTable$column_width()`: Width of a column in cells.

Usage:

```
DataTable$column_width(column)
```

Arguments:

`column` Column name or number.

`DataTable$set_column_width()`: Set a column's width (at least 1 cell). Sends `"datatable.column_resized"`.

Usage:

```
DataTable$set_column_width(column, width)
```

Arguments:

`column` Column name or number.

`width` Width in cells.

`DataTable$auto_size_column()`: Fit a column to its content. Rows are sampled, never all scanned unless asked for, so this is fast for huge tables.

Usage:

```
DataTable$auto_size_column(column, rows = c("sample", "visible", "all"))
```

Arguments:

`column` Column name or number.

`rows` `"sample"` (the visible rows plus the first and last 500; default), `"visible"` (visible rows only) or `"all"` (every row of the view; slow for millions of rows).

`DataTable$auto_size_all()`: Fit every visible column to its content.

Usage:

```
DataTable$auto_size_all(rows = c("sample", "visible", "all"))
```

Arguments:

`rows` See `auto_size_column()`.

`DataTable$set_column_visible()`: Show or hide a column.

Usage:

```
DataTable$set_column_visible(column, visible = TRUE)
```

Arguments:

`column` Column name or number.

`visible` `TRUE` to show, `FALSE` to hide.

`DataTable$column_info()`: Describe the columns: name, label, width, visible, sortable.

Usage:

```
DataTable$column_info()
```

`DataTable$find()`: Search the formatted cell text and move the cursor to the first match at or after the cursor (reading order, wrapping around). Large tables are scanned in chunks; the data is never converted to one big string matrix.

Usage:

```
DataTable$find(query, column = NULL, case_sensitive = FALSE, regex = FALSE)
```

Arguments:

`query` Text or regular expression (PCRE).

`column` Search only this column (name or number); `NULL`: all visible columns.

`case_sensitive` Match case?

`regex` Is `query` a regular expression?

Returns: `TRUE` if there is a match (invisibly).

`DataTable$find_next()`: Go to the next match.

Usage:

```
DataTable$find_next()
```

`DataTable$find_previous()`: Go to the previous match.

Usage:

```
DataTable$find_previous()
```

`DataTable$clear_find()`: Forget the search and close the find bar.

Usage:

```
DataTable$clear_find()
```

`DataTable$edit_cell()`: Edit a cell in a small dialog (needs `editable = TRUE`). The default is the cursor cell.

Usage:

```
DataTable$edit_cell(position = NULL, column = NULL)
```

Arguments:

`position` Position in the view (default: the cursor row).

`column` Column name or number (default: the cursor / active column).

`DataTable$set_cell()`: Change one cell of the table's own copy of the data (the data frame you passed in is never modified; read the result with `$data`). Runs `on_edit`, then sends `"datatable.cell_changed"` with `row`, `column`, `old` and `value`.

Usage:

```
DataTable$set_cell(row, column, value)
```

Arguments:

`row` Original row number.

`column` Column name or number.

`value` New value (must fit the column type).

`DataTable$action_cursor_up()`: Actions used by the bindings.

Usage:

```
DataTable$action_cursor_up()
```

`DataTable$action_cursor_down()`: Move down.

Usage:

```
DataTable$action_cursor_down()
```

`DataTable$action_extend_selection_up()`: Extend selection upward.

Usage:

```
DataTable$action_extend_selection_up()
```

`DataTable$action_extend_selection_down()`: Extend selection downward.

Usage:

```
DataTable$action_extend_selection_down()
```

`DataTable$action_move_column_left()`: Move the active column left in display order.

Usage:

```
DataTable$action_move_column_left()
```

`DataTable$action_move_column_right()`: Move the active column right in display order.

Usage:

```
DataTable$action_move_column_right()
```

`DataTable$action_copy_selection()`: Copy selected rows.

Usage:

```
DataTable$action_copy_selection()
```

`DataTable$action_cursor_left()`: Move left (cell cursor) or scroll left.

Usage:

```
DataTable$action_cursor_left()
```

`DataTable$action_cursor_right()`: Move right (cell cursor) or scroll right.

Usage:

```
DataTable$action_cursor_right()
```

`DataTable$action_page_up()`: Move one page up.

Usage:

```
DataTable$action_page_up()
```

`DataTable$action_page_down()`: Move one page down.

Usage:

```
DataTable$action_page_down()
```

`DataTable$action_first_row()`: Go to the first row.

Usage:

```
DataTable$action_first_row()
```

`DataTable$action_last_row()`: Go to the last row.

Usage:

```
DataTable$action_last_row()
```

`DataTable$action_open_find()`: Open the find bar (Ctrl+F).

Usage:

```
DataTable$action_open_find()
```

`DataTable$action_open_goto()`: Open the "go to row" bar (Ctrl+G).

Usage:

```
DataTable$action_open_goto()
```

`DataTable$action_find_next()`: Go to the next match (F3).

Usage:

```
DataTable$action_find_next()
```

`DataTable$action_find_previous()`: Go to the previous match (Shift+F3).

Usage:

```
DataTable$action_find_previous()
```

`DataTable$action_widen_column()`: Make the active column one cell wider (Alt+Right).

Usage:

```
DataTable$action_widen_column()
```

`DataTable$action_narrow_column()`: Make the active column one cell narrower (Alt+Left).

Usage:

```
DataTable$action_narrow_column()
```

`DataTable$action_sort_column()`: Sort by the active column (S): ascending, descending, off.

Usage:

```
DataTable$action_sort_column()
```

`DataTable$action_activate()`: Send `"datatable.row_activated"` for the cursor row (or edit the cell when the table is editable).

Usage:

```
DataTable$action_activate()
```

`DataTable$on_mouse_down()`: Mouse: clicking a row moves the cursor, clicking the header sends `"datatable.header_selected"` (and sorts when `header_sort` is on); dragging a column separator in the header resizes the column; a double click edits a cell of an editable table.

Usage:

```
DataTable$on_mouse_down(event)
```

Arguments:

`event` A `MouseEvent`.

`DataTable$on_drag_move()`: Resizing a column by dragging its header separator.

Usage:

```
DataTable$on_drag_move(event)
```

Arguments:

`event` A `MouseEvent`.

`DataTable$on_drag_end()`: Ends a column resize.

Usage:

```
DataTable$on_drag_end(event)
```

Arguments:

`event` A `MouseEvent`.

`DataTable$on_key()`: Keys for the find / go-to bar while it is open.

Usage:

```
DataTable$on_key(event)
```

Arguments:

`event` A `KeyEvent`.

`DataTable$on_paste()`: Pasted text goes into the find / go-to bar when open.

Usage:

```
DataTable$on_paste(event)
```

Arguments:

`event` A `PasteEvent`.

`DataTable$on_mouse_scroll()`: The mouse wheel scrolls three rows.

Usage:

```
DataTable$on_mouse_scroll(event)
```

Arguments:

`event` A `MouseEvent`.

`DataTable$content_width()`: Natural width: all columns.

Usage:

```
DataTable$content_width()
```

`DataTable$content_height()`: Natural height: header plus rows (at most 1000).

Usage:

```
DataTable$content_height(width)
```

Arguments:

`width` Content width.

`DataTable$paint()`: Draw the visible part of the table.

Usage:

```
DataTable$paint(buffer, area, st)
```

Arguments:

`buffer` A ScreenBuffer.
`area` Visible part of the region.
`st` Computed style.

`DataTable$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`data_profile` *Compact vector profile*

Description

Summarizes numeric/integer, character/factor, logical, Date, and POSIXct vectors. Numeric summaries use base R type-7 quartiles and sample standard deviation (`stats::sd`). Numeric data also gets an eight-bin histogram rendered with the existing sparkline widget. Without `sample`, vector summaries analyze the full in-memory vector. Character top-value counts cap their frequency table at 100,000 analyzed non-missing values. Supplying `sample` analyzes deterministic, evenly spaced positions. A `table_source` is rejected to prevent an implicit full scan; fetch or sample data explicitly first. `data` may also be a reactive function.

Usage

```
data_profile(
  data,
  name = NULL,
  sample = NULL,
  top_n = 5L,
  id = NULL,
  classes = NULL,
  style = NULL,
  disabled = FALSE
)
```

Arguments

`data`	An atomic vector, factor, Date, or POSIXct vector.
`name`	Heading for the profile.
`sample`	Optional positive number of evenly spaced values to analyze.
`top_n`	Number of frequent character/factor values to display.
`id`	Optional identifier.
`classes`	Optional classes.
`style`	A `style()`.
`disabled`	Disable the widget?

Value

A `DataProfile` widget with reactive `$data` and `$summary`.

Examples

```
data_profile(mtcars$mpg, name = "mpg")
```

`data_table` *Data table*

Description

A fast, read-only viewer for data frames: a header, optional row names, automatic column widths, keyboard and mouse navigation, horizontal and vertical scrolling, and a row or cell cursor. Only the visible rows are formatted and painted, so tables with millions of rows scroll smoothly.

Usage

```
data_table(
  data,
  cursor = "row",
  columns = NULL,
  formatters = NULL,
  row_names = NA,
  header = TRUE,
  zebra = FALSE,
  row_style = NULL,
  cell_style = NULL,
  max_column_width = 40L,
  frozen_columns = 0L,
  header_sort = FALSE,
  editable = FALSE,
  on_edit = NULL,
  id = NULL,
  classes = NULL,
  style = NULL
)
```

Arguments

`data`	A data frame (tibbles work too) or a matrix.
`cursor`	`"row"` (default), `"cell"` or `"none"`.
`columns`	Named list of `column()` options, e.g. `list(score = column(width = 10, align = "right"))`.
`formatters`	Named list of `function(x)` returning strings, e.g. `list(score = function(x) sprintf("%.2f", x))`.
`row_names`	Show row names? `NA` (default) shows them only when they are not the automatic `1..n` (e.g. `mtcars`).
`header`	Show the header row?
`zebra`	Stripe alternate rows?
`row_style`	`function(row, data)` returning a `style()` (or `NULL`) for a data row.
`cell_style`	`function(value, row, column)` returning a `style()` (or `NULL`) for a cell.

`max_column_width`
: Upper limit for automatic column widths.

`frozen_columns`
: Number of leading columns that stay in place while the others scroll horizontally.

`header_sort`
: Sort when a header is clicked (again: descending, then off; Shift+click adds a secondary key)?

`editable`
: Allow editing cells (Enter or double click opens a small dialog)? The table edits its own copy of the data; read it with `$data` or react to `"datatable.cell_changed"`. **Experimental.**

`on_edit`
: `function(row, column, value)` called before an edit is applied; return `NULL`/`TRUE` to accept or a message to reject.

`id`
: Optional identifier.

`classes`
: Optional classes.

`style`
: A `style()`; by default the table fills its parent.

Details

Keys: arrows move the cursor (with a row cursor, Left/Right scroll the columns), Page Up/Down move a page, Home/End go to the first / last row, Enter activates the row (or edits the cell when `editable`), Ctrl+F finds text (Enter/Down next, Up previous, Tab all columns / this column, Ctrl+R regex, Alt+C case, Esc close), F3 / Shift+F3 next / previous match, Ctrl+G jumps to a row, `S` (with `header_sort`) sorts by the active column (ascending, descending, off), Alt+Left / Alt+Right narrow / widen the active column. Clicking a row selects it; clicking the header sends `"datatable.header_selected"` (and sorts with `header_sort`); dragging the separator between two header cells resizes a column; the wheel scrolls. The header and the first `frozen_columns` columns stay in place while the rest scrolls.

Messages (`event$data`):

- `"datatable.row_selected"` / `"datatable.cell_selected"`: the cursor moved; `row` (original row number), `position` (in the view), `value` (the row as a list, or the cell value), `column` (cell cursor);
- `"datatable.row_activated"`: Enter; same data;
- `"datatable.header_selected"`: `column` (name) and `index`;
- `"datatable.sorted"` (`columns`, `decreasing`), `"datatable.column_resized"` (`column`, `width`), `"datatable.columns_changed"` (`column`, `visible`), `"datatable.found"` (`position`, `row`, `column`), `"datatable.cell_changed"` (`row`, `column`, `old`, `value`).

Methods: `set_data()`, `sort(by, decreasing)` (several columns allowed), `clear_sort()`, `toggle_sort()`, `filter(rows)`, `move_cursor(row, column)`, `goto_row()`, `scroll_to_row()`, `selected_row()`, `row_data(row)`, `set_column_width()`, `auto_size_column()`, `auto_size_all()`, `set_column_visible()`, `column_info()`, `find()`, `find_next()`, `find_previous()`, `edit_cell()`, `set_cell()`.

Value

A `DataTable` widget.

Examples

```
tbl <- data_table(mtcars, id = "cars", zebra = TRUE,
                  formatters = list(mpg = function(x) sprintf("%.1f", x)))
render_widget(tbl, 60, 6)
```

`db_connection` *Wrap a DBI connection for use by termr SQL widgets*

Description

Wrap a DBI connection for use by termr SQL widgets

Usage

```
db_connection(con, name = NULL, owned = FALSE)
```

Arguments

`con`	A DBI connection.
`name`	Optional connection label.
`owned`	Disconnect this connection when `disconnect()` is called? Defaults to `FALSE` for externally managed connections.

Value

A lightweight connection wrapper with `query()`, `execute()`, `is_valid()`, and `disconnect()` methods.

`db_explorer` *Database explorer widget*

Description

Composes a lazy object tree, SQLite table preview, SQL editor and result table, details and in-memory query history. The object tree loads metadata only when its Tables or Views node is expanded. DBI/RSQLite remain optional dependencies.

Usage

```
db_explorer(connection, id = NULL)
```

Arguments

`connection`	A `db_connection()` wrapper or open DBI connection. Raw connections are treated as externally owned.
`id`	Optional widget identifier.

Value

A `DatabaseExplorer` widget with `$refresh()`, `$run_query()`, `$open_table()`, and `$query_history()` methods.

Examples

```
if (requireNamespace("DBI", quietly = TRUE) && requireNamespace("RSQLite", quietly = TRUE)) {
  con <- DBI::dbConnect(RSQLite::SQLite(), ":memory:")
  DBI::dbWriteTable(con, "mtcars", mtcars)
  pilot <- test_app(app(db_explorer(db_connection(con, owned = TRUE))))
  pilot$stop() # App teardown closes its owned connection.
}
```

---

db_metadata *Create a DBI metadata interface*

Description

The returned functions use DBI's portable table and field listing. SQLite additionally exposes views from its catalog. Driver packages can use this small callback protocol when building their own metadata views; the explorer does not yet have an adapter registry.

Usage

```
db_metadata(connection)
```

Arguments

`connection`	A `db_connection()` wrapper or open DBI connection.

Value

A list with `schemas()`, `tables(schema)`, `views(schema)`, `columns(table, schema)`, and `quote(identifier)` functions.

---

db_query_source *Create a lazy DBI query source*

Description

The source implements `table_source()` and fetches bounded pages for a query result. The built-in pager supports RSQLite only. Other DBI drivers can provide an `adapter` list with `column_names(connection, query, params)`, `row_count(connection, query, params)`, and `get_rows(connection, query, params, start, count, columns)` callbacks. This keeps dialect-specific pagination in the backend adapter rather than assuming all DBI drivers use the same SQL.

Usage

```
db_query_source(connection, query, params = NULL, adapter = NULL)
```

Arguments

connection	A db_connection() wrapper or open DBI connection. It remains owned by its caller; closing this source never disconnects it.
query	A single query that returns a result set.
params	Optional DBI parameter list passed to the query.
adapter	Optional backend adapter callbacks. NULL selects the built-in RSQLite pager when connection is an RSQLite connection.

Details

The built-in SQLite adapter wraps the query as a subquery, runs one cached COUNT(*), and adds bound LIMIT/OFFSET values to page requests. A single trailing semicolon is removed before wrapping. The query must produce one result set; arbitrary SQL statements are not parsed or rewritten. The row count can be expensive, but it is computed once at source creation and on explicit refresh, never on repaint.

The returned source is read-only. Sorting, filtering, and search are not pushed into the user query.

Value

An experimental table_source() accepted directly by data_table().

---

db_table_source *Create a lazy source for a SQLite table*

Description

This optional adapter uses bounded LIMIT/OFFSET queries and parameterized filter values. It supports equals, contains, range, and missing table_filter() filters and sorting by quoted column names. Arbitrary queries and other SQL dialects are intentionally outside this adapter; custom table_source() callbacks can implement driver-specific pagination.

Usage

```
db_table_source(connection, table, owned = FALSE)
```

Arguments

connection	An open RSQLite connection, or a db_connection() wrapper of one.
table	A table name or DBI::Id.
owned	If TRUE, source$close() disconnects this connection. The default leaves connection ownership with the caller.

Value

An experimental table source accepted by `data_table()`.

`diff_screen` *Compare two screen buffers*

Description

Computes the minimal set of horizontal runs of cells that must be rewritten to turn the screen `old` into `new`. Nearby runs on the same row are merged when rewriting the unchanged cells between them is cheaper than an extra cursor movement.

Usage

```
diff_screen(old, new, merge_gap = 3L)
```

Arguments

`old`, `new`	ScreenBuffer objects (`old` may be `NULL`).
`merge_gap`	Runs separated by at most this many unchanged cells are merged.

Details

If `old` is `NULL` or has different dimensions, the patch covers every non-blank cell of `new` and is marked `full = TRUE`; the renderer then clears the screen first.

Value

A `termr_patch`: a list with `full` (logical), `width`, `height` and `runs`, a list of runs. Each run has `y`, `x` and vectors `chars`, `fg`, `bg`, `attrs` copied from `new`.

Examples

```
a <- screen_buffer(10, 2)
b <- a$copy()
b$put_text(3, 2, "hi")
diff_screen(a, b)
```

dropdown *Drop-down select*

Description

Shows the selected choice; Enter, Space, Down or a click opens a list below it. Choosing an option sends "dropdown.changed" (event$data: value, label); Escape or a click outside closes the list.

Usage

```
dropdown(
  choices,
  value = NULL,
  prompt = "Select",
  id = NULL,
  classes = NULL,
  style = NULL,
  disabled = FALSE
)
```

Arguments

choices	A character vector; names, if present, are shown.
value	Initially selected value, or NULL.
prompt	Text shown when nothing is selected.
id	Optional identifier.
classes	Optional classes.
style	A style().
disabled	Disabled selects cannot be opened.

Details

Named dropdown() rather than select() so that it does not mask dplyr::select(); the widget type (for selectors) is Select.

Value

A Select widget with a reactive value.

Examples

```
dropdown(c("Comma separated" = "csv", "JSON" = "json"), value = "csv", id = "format")
```

`Event` *Events*

Description

Everything that happens in a termr application is an event: key presses, terminal resizes, focus changes, widgets being mounted, and messages that widgets send (such as `"button.pressed"`).

Events are delivered to their *target* widget and, if they bubble, to each ancestor in turn and finally to the App. Any handler can call `event$stop()` to stop propagation.

Event types and their classes:

- `"key"`: `KeyEvent` with fields `key`, `char`, `ctrl`, `alt`, `shift`;
- `"resize"`: `ResizeEvent` with `width` and `height`;
- `"paste"`: `PasteEvent` with the pasted `text` (bracketed paste);
- `"focus"` / `"blur"`: `FocusEvent` / `BlurEvent` (do not bubble);
- `"mount"` / `"unmount"`: `MountEvent` / `UnmountEvent` (do not bubble);
- any other name: `MessageEvent`, created by `widget$post_message()`.

Public fields

`type` Event type, e.g. `"key"` or `"button.pressed"`.

`sender` The widget that created the event (or `NULL`).

`target` The widget the event is delivered to first.

`current` The widget currently handling the event.

`data` A list of event-specific values.

`bubbles` Is the event passed on to ancestors?

`time` Creation time (seconds, monotonic).

Active bindings

`stopped` Has propagation been stopped?

`default_prevented` Was `prevent_default()` called?

Methods

Public methods:

- `Event$new()`
- `Event$stop()`
- `Event$prevent_default()`
- `Event$format()`
- `Event$print()`
- `Event$clone()`

`Event$new()`: Create an event.

Usage:

```
Event$new(type, sender = NULL, data = list(), bubbles = TRUE)
```

Arguments:

`type` Event type.

`sender` Sending widget.

`data` List of values.

`bubbles` Does the event bubble?

`Event$stop()`: Stop the event from reaching further handlers.

Usage:

```
Event$stop()
```

`Event$prevent_default()`: Ask the sender not to perform its default behaviour.

Usage:

```
Event$prevent_default()
```

`Event$format()`: One-line description.

Usage:

```
Event$format(...)
```

Arguments:

`...` Ignored.

`Event$print()`: Print the event.

Usage:

```
Event$print(...)
```

Arguments:

`...` Ignored.

`Event$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Super class

`Event` -> `KeyEvent`

Public fields

`key` Canonical key name, e.g. `"a"`, `"enter"`, `"ctrl+c"`.

`char` The printable character, or `""`.

`ctrl`, `alt`, `shift` Modifier flags.

Methods

Public methods:

- KeyEvent$new()
- KeyEvent$is_printable()
- KeyEvent$clone()

KeyEvent$new(): Create a key event.

Usage:

```
KeyEvent$new(key, char = NULL)
```

Arguments:

key Key name (normalised).

char Printable character, if any.

KeyEvent$is_printable(): Is this a printable character without ctrl/alt?

Usage:

```
KeyEvent$is_printable()
```

KeyEvent$clone(): The objects of this class are cloneable with this method.

Usage:

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

Arguments:

deep Whether to make a deep clone.

Super class

Event -> ResizeEvent

Public fields

width, height New terminal size.

Methods

Public methods:

- ResizeEvent$new()
- ResizeEvent$clone()

ResizeEvent$new(): Create a resize event.

Usage:

```
ResizeEvent$new(width, height)
```

Arguments:

width, height Terminal size.

ResizeEvent$clone(): The objects of this class are cloneable with this method.

Usage:

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

Arguments:

deep Whether to make a deep clone.

Super class

Event -> PasteEvent

Public fields

`text` The pasted text, normalised: line breaks are LF, tabs are kept and other control characters are removed.

Methods

Public methods:

- PasteEvent$new()
- PasteEvent$clone()

`PasteEvent$new()`: Create a paste event.

Usage:

```
PasteEvent$new(text)
```

Arguments:

`text` Pasted text.

`PasteEvent$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Super class

Event -> FocusEvent

Methods

Public methods:

- FocusEvent$new()
- FocusEvent$clone()

`FocusEvent$new()`: Create a focus event.

Usage:

```
FocusEvent$new(widget)
```

Arguments:

`widget` The widget that received focus.

`FocusEvent$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Super class

Event -> BlurEvent

Methods

Public methods:

- BlurEvent$new()
- BlurEvent$clone()

BlurEvent$new(): Create a blur event.

Usage:

```
BlurEvent$new(widget)
```

Arguments:

widget The widget that lost focus.

BlurEvent$clone(): The objects of this class are cloneable with this method.

Usage:

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

Arguments:

deep Whether to make a deep clone.

Super class

Event -> MountEvent

Methods

Public methods:

- MountEvent$new()
- MountEvent$clone()

MountEvent$new(): Create a mount event.

Usage:

```
MountEvent$new(widget)
```

Arguments:

widget The mounted widget.

MountEvent$clone(): The objects of this class are cloneable with this method.

Usage:

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

Arguments:

deep Whether to make a deep clone.

Super class

Event -> UnmountEvent

Methods

Public methods:

- UnmountEvent$new()
- UnmountEvent$clone()

`UnmountEvent$new()`: Create an unmount event.

Usage:

```
UnmountEvent$new(widget)
```

Arguments:

`widget` The removed widget.

`UnmountEvent$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Super class

Event -> MessageEvent

Methods

Public methods:

- MessageEvent$new()
- MessageEvent$clone()

`MessageEvent$new()`: Create a custom message.

Usage:

```
MessageEvent$new(type, sender = NULL, data = list(), bubbles = TRUE)
```

Arguments:

`type` Message name, e.g. `"counter.changed"`.

`sender` Sending widget.

`data` List of values.

`bubbles` Does the message bubble?

`MessageEvent$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Super class

Event -> MouseEvent

Public fields

action "down", "up", "move", "scroll" or "click".

button "left", "middle", "right" or "none".

direction Wheel direction for scroll events, else NA.

screen_x, screen_y Pointer position on the screen (1-based).

x, y Pointer position inside the target widget (1-based, relative to the widget's region).

origin_x, origin_y Where the button went down (screen coordinates) for drag.* events, else NA.

shift, ctrl, alt Modifier flags.

Methods

Public methods:

- MouseEvent$new()
- MouseEvent$offset_in()
- MouseEvent$clone()

MouseEvent$new(): Create a mouse event.

Usage:

```
MouseEvent$new(
  action,
  screen_x,
  screen_y,
  button = "none",
  direction = NA_character_,
  shift = FALSE,
  ctrl = FALSE,
  alt = FALSE
)
```

Arguments:

action Event action.

screen_x, screen_y Screen position.

button Button name.

direction Wheel direction (scroll events).

shift, ctrl, alt Modifiers.

MouseEvent$offset_in(): The pointer position relative to another widget.

Usage:

```
MouseEvent$offset_in(widget)
```

Arguments:

`widget` A widget with a region.

Returns: `c(x =, y =)`, 1-based; may be outside the widget.

`MouseEvent$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`Grid` *Grid container*

Description

Arranges children in a grid. See `grid_layout()`.

Super class

`Widget` -> `Grid`

Methods

Public methods:

- `Grid$default_style()`
- `Grid$clone()`

`Grid$default_style()`: The built-in style.

Usage:

```
Grid$default_style()
```

`Grid$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

grid_layout *Grid container*

Description

Places children in a grid, row by row. Column (and optionally row) tracks are sized like widgets: a number of cells, `"auto"` (natural size of the content), `"<n>fr"` (share of the remaining space) or `"<n>%"`. Rows not listed are `"auto"`. A child can cover several cells with `style(column_span = 2, row_span = 2)`.

Usage

```
grid_layout(
  ...,
  columns = 2L,
  rows = NULL,
  gap = 0L,
  id = NULL,
  classes = NULL,
  style = NULL
)
```

Arguments

`...`	Child widgets.
`columns`	A number of equal columns, or a vector of column sizes.
`rows`	Optional vector of row sizes.
`gap`	Space between cells: one value, or `c(rows, columns)`.
`id`	Optional identifier.
`classes`	Optional classes.
`style`	A `style()`.

Details

Named `grid_layout()` because `grid()` would mask `graphics::grid()`.

Value

A `Grid` widget.

Examples

```
ui <- grid_layout(
  label("a"), label("b"), label("c"),
  label("wide", style = style(column_span = 2)),
  columns = c("1fr", "2fr"), gap = 1
)
render_widget(ui, 20, 5)
```

`Horizontal` *Horizontal container*

Description

Arranges children left to right. See `horizontal()`.

Super class

`Widget` -> `Horizontal`

Methods

Public methods:

- `Horizontal$default_style()`
- `Horizontal$clone()`

`Horizontal$default_style()`: The built-in style.

Usage:

```
Horizontal$default_style()
```

`Horizontal$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`Input` *Input widget*

Description

A single-line text input. See `input()`.

Super class

`Widget` -> `Input`

Public fields

`paint_states` Cursor and selection changes only repaint.

`focusable` Inputs can be focused.

`placeholder` Text shown while the input is empty.

`password` Show bullets instead of the characters?

`max_length` Maximum number of characters, or `NULL`.

`validate` `NULL` or `function(value)` returning `NULL` (valid) or an error message.

Active bindings

`value` The text (reactive). Assigning moves the cursor to the end and sends `"input.changed"`.

`cursor_position` Cursor position in characters (0 = start).

`selection` The selected text (`""` if none).

`valid` Does the value pass `validate`?

`error` The validation message, or `NULL`.

Methods

Public methods:

- `Input$new()`
- `Input$default_style()`
- `Input$pseudo_states()`
- `Input$default_bindings()`
- `Input$on_key()`
- `Input$on_mouse_down()`
- `Input$on_mouse_move()`
- `Input$on_paste()`
- `Input$insert()`
- `Input$clear()`
- `Input$select_range()`
- `Input$copy()`

- Input$action_cursor_left()
- Input$action_cursor_right()
- Input$action_cursor_home()
- Input$action_cursor_end()
- Input$action_word_left()
- Input$action_word_right()
- Input$action_select_left()
- Input$action_select_right()
- Input$action_select_home()
- Input$action_select_end()
- Input$action_select_word_left()
- Input$action_select_word_right()
- Input$action_select_all()
- Input$action_delete_left()
- Input$action_delete_right()
- Input$action_delete_word_left()
- Input$action_delete_word_right()
- Input$action_delete_to_start()
- Input$action_delete_to_end()
- Input$action_paste()
- Input$action_submit()
- Input$content_width()
- Input$render_lines()
- Input$clone()

`Input$new()`: Create an input.

Usage:

```
Input$new(
  value = "",
  placeholder = "",
  id = NULL,
  classes = NULL,
  style = NULL,
  disabled = FALSE,
  password = FALSE,
  max_length = NULL,
  validate = NULL
)
```

Arguments:

`value` Initial text.

`placeholder` Placeholder text.

`id`, `classes`, `style`, `disabled` See Widget.

`password` Mask the characters?

`max_length` Maximum number of characters.

`validate` Validation function.

`Input$default_style()`: The built-in style.

Usage:

```
Input$default_style()
```

`Input$pseudo_states()`: Active states: focus, hover, disabled and invalid.

Usage:

```
Input$pseudo_states()
```

`Input$default_bindings()`: Editing key bindings.

Usage:

```
Input$default_bindings()
```

`Input$on_key()`: Printable keys replace the selection or are inserted at the cursor. Ctrl+C / Ctrl+X copy / cut when text is selected (otherwise Ctrl+C keeps its app meaning).

Usage:

```
Input$on_key(event)
```

Arguments:

`event` A `KeyEvent`.

`Input$on_mouse_down()`: Clicking places the cursor; shift+click and dragging select.

Usage:

```
Input$on_mouse_down(event)
```

Arguments:

`event` A `MouseEvent`.

`Input$on_mouse_move()`: Dragging with the left button extends the selection.

Usage:

```
Input$on_mouse_move(event)
```

Arguments:

`event` A `MouseEvent`.

`Input$on_paste()`: Pasted text is inserted as one edit. An Input is a single line, so line breaks in the paste become spaces.

Usage:

```
Input$on_paste(event)
```

Arguments:

`event` A `PasteEvent`.

`Input$insert()`: Insert text at the cursor (replacing the selection).

Usage:

```
Input$insert(text)
```

Arguments:

`text` A string.

`Input$clear()`: Remove all text.

Usage:

```
Input$clear()
```

`Input$select_range()`: Select a range of characters.

Usage:

```
Input$select_range(start, end)
```

Arguments:

`start, end` Positions (0 = before the first character).

`Input$copy()`: Copy the selection to the app clipboard (and, when `system = TRUE` and supported, to the system clipboard via OSC 52).

Usage:

```
Input$copy(system = TRUE)
```

Arguments:

`system` Also write to the system clipboard?

`Input$action_cursor_left()`: Move one character left.

Usage:

```
Input$action_cursor_left()
```

`Input$action_cursor_right()`: Move one character right.

Usage:

```
Input$action_cursor_right()
```

`Input$action_cursor_home()`: Move to the start.

Usage:

```
Input$action_cursor_home()
```

`Input$action_cursor_end()`: Move to the end.

Usage:

```
Input$action_cursor_end()
```

`Input$action_word_left()`: Move to the previous word.

Usage:

```
Input$action_word_left()
```

`Input$action_word_right()`: Move to the next word.

Usage:

```
Input$action_word_right()
```

`Input$action_select_left()`: Extend the selection left.

Usage:

```
Input$action_select_left()
```

`Input$action_select_right()`: Extend the selection right.

Usage:

```
Input$action_select_right()
```

`Input$action_select_home()`: Select to the start.

Usage:

```
Input$action_select_home()
```

`Input$action_select_end()`: Select to the end.

Usage:

```
Input$action_select_end()
```

`Input$action_select_word_left()`: Select to the previous word.

Usage:

```
Input$action_select_word_left()
```

`Input$action_select_word_right()`: Select to the next word.

Usage:

```
Input$action_select_word_right()
```

`Input$action_select_all()`: Select everything.

Usage:

```
Input$action_select_all()
```

`Input$action_delete_left()`: Delete the selection or the character before the cursor.

Usage:

```
Input$action_delete_left()
```

`Input$action_delete_right()`: Delete the selection or the character at the cursor.

Usage:

```
Input$action_delete_right()
```

`Input$action_delete_word_left()`: Delete the word before the cursor.

Usage:

```
Input$action_delete_word_left()
```

`Input$action_delete_word_right()`: Delete the word after the cursor.

Usage:

```
Input$action_delete_word_right()
```

`Input$action_delete_to_start()`: Delete everything before the cursor.

Usage:

```
Input$action_delete_to_start()
```

`Input$action_delete_to_end()`: Delete everything after the cursor.

Usage:

```
Input$action_delete_to_end()
```

`Input$action_paste()`: Insert the app clipboard.

Usage:

```
Input$action_paste()
```

`Input$action_submit()`: Send an `"input.submitted"` message (Enter).

Usage:

```
Input$action_submit()
```

`Input$content_width()`: Natural content width.

Usage:

```
Input$content_width()
```

`Input$render_lines()`: The visible part of the text, scrolled so the cursor is visible, with the cursor drawn in reverse video and the selection highlighted when focused.

Usage:

```
Input$render_lines(width = NA_integer_)
```

Arguments:

`width` Content width.

`Input$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`input` *Single-line text input*

Description

A focusable text field.

Usage

```
input(
  value = "",
  placeholder = "",
  id = NULL,
  classes = NULL,
  style = NULL,
  disabled = FALSE,
  password = FALSE,
  max_length = NULL,
  validate = NULL
)
```

Arguments

value	Initial text.
placeholder	Text shown while the input is empty.
id	Optional identifier (for #id selectors).
classes	Optional classes (for .class selectors).
style	A style().
disabled	Disabled inputs cannot be focused or edited.
password	Show bullets instead of the typed characters.
max_length	Maximum number of characters.
validate	Optional function(value) returning NULL (or TRUE) when the value is valid, or an error message.

Details

Keys: typing inserts at the cursor (replacing a selection); Left/Right/Home/End move (Ctrl+E: end); Ctrl+Left/Right move by word; Shift (+Ctrl) with those keys selects; Ctrl+A selects all; Backspace/Delete delete a character or the selection; Ctrl+Backspace / Alt+Backspace / Ctrl+W and Ctrl+Delete / Alt+D delete a word; Ctrl+U / Ctrl+K delete to the start / end; Ctrl+C, Ctrl+X and Ctrl+V copy, cut and paste with the app clipboard (app$clipboard; Ctrl+C without a selection keeps its app meaning, quit by default). The mouse places the cursor; shift+click and dragging select.

Messages (event$data): "input.changed" (value, valid) on every change, "input.submitted" (value, valid) on Enter, and "input.valid" / "input.invalid" (value, error) when the result of validate changes. Invalid inputs have the :invalid state (red border by default).

Value

An Input widget. Read or set the text with $value; see also $valid, $error, $selection, $select_range().

Examples

```
name <- input(placeholder = "Your name", id = "name",
              validate = function(x) if (!nzchar(x)) "Required")
name$valid
name$value <- "Ada"
name$valid
```

`inspect_signal` *Inspect one node in the experimental signal graph*

Description

Return dependency and subscriber names, lifecycle state, and cumulative counters for a signal, computed value, or watcher.

Usage

```
inspect_signal(x)
```

Arguments

`x`	A `signal()`, `computed()`, or `watch()` result.

Value

A list describing the node and its direct graph edges.

`inspect_widget` *Inspect a widget*

Description

Collects what termr knows about a widget: type, id, classes, region, states, reactive state values, computed style, matching stylesheet rules, bindings, parent and children. Printing the result gives a readable report.

Usage

```
inspect_widget(x, selector = NULL)
```

Arguments

`x`	An App or a widget.
`selector`	A selector to find the widget when `x` is an app (or a container).

Value

A list of class `termr_inspection`.

Examples

```
ui <- vertical(button("Save", id = "save", classes = "primary"))
inspect_widget(ui, "#save")
```

`json_view` *Collapsible viewer for R list-like data*

Description

Displays already-parsed R objects as a `tree_view()`. Named lists and data frames are objects; unnamed lists and vectors are arrays. Children are created when a node is expanded. At most `page_size` children are created per page; a `Load next ...` node exposes the next page. Character scalars are quoted, logical values use lowercase JSON spelling, `NULL` is shown as `null`, and R `NA` is shown as `NA` to keep it distinct from JSON `null`. Set `parse = TRUE` for JSON text when optional package `jsonlite` is installed; this is a convenience over the primary R-object interface.

Usage

```
json_view(
  x,
  parse = FALSE,
  page_size = 100L,
  id = NULL,
  classes = NULL,
  style = NULL,
  disabled = FALSE
)
```

Arguments

`x`	An R object, or one JSON string when `parse = TRUE`.
`parse`	Parse a JSON character string with optional `jsonlite`?
`page_size`	Maximum children created when a tree node is expanded.
`id`	Optional identifier.
`classes`	Optional classes.
`style`	A `style()`.
`disabled`	Disable focus and tree interaction?

Value

A `TreeView` with lazily loaded object/array children.

Examples

```
json_view(list(user = list(id = 42, name = "Alice"), active = TRUE))
```

KeyValue *KeyValue widget*

Description

See `key_value()`.

Super class

`Widget` -> `KeyValue`

Active bindings

`data` Named list of values (reactive).

Methods

Public methods:

- `KeyValue$new()`
- `KeyValue$default_style()`
- `KeyValue$render()`
- `KeyValue$clone()`

`KeyValue$new()`: Create a key/value list. See `key_value()`.

Usage:

```
KeyValue$new(data = list(), id = NULL, classes = NULL, style = NULL)
```

Arguments:

`data` Named list or vector.

`id`, `classes`, `style` See Widget.

`KeyValue$default_style()`: The built-in style.

Usage:

```
KeyValue$default_style()
```

`KeyValue$render()`: Aligned key and value lines.

Usage:

```
KeyValue$render()
```

`KeyValue$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

key_event *Create a key event*

Description

Mostly useful in tests: see `test_app()` for simulating key presses.

Usage

```
key_event(key)
```

Arguments

key	Key name such as `"a"`, `"enter"`, `"ctrl+c"` or `"shift+tab"`.

Value

A `KeyEvent`.

Examples

```
key_event("ctrl+c")
```

key_value *Key/value list*

Description

Shows names and values in two aligned columns, e.g. a summary of a data set.

Usage

```
key_value(data = list(), id = NULL, classes = NULL, style = NULL)
```

Arguments

data	A named list or vector.
id	Optional identifier.
classes	Optional classes.
style	A `style()`.

Value

A `KeyValue` widget with a reactive `data` field.

Choosing a key/value widget

- key_value(): a few known values shown in full, such as a summary panel. Not focusable, does not scroll; values are not wrapped.
- property_grid(): an inspector for many or arbitrary named values. Focusable and scrollable, wraps long values, formats vectors and lists compactly and accepts a format function.
- record_view(): property_grid() for exactly one record, such as the selected row of a data_table(); accepts a one-row data frame.

Examples

```
render_widget(key_value(list(Rows = 32, Columns = 11, Source = "mtcars")), 20, 3)
```

---

knit_termr *Render a static representation for knitr or Quarto*

Description

Render a static representation for knitr or Quarto

Usage

```
knit_termr(x, width = 80, height = 24, format = c("markdown", "html"))
```

Arguments

x	A widget, Screen, App, or ScreenBuffer.
width, height	Frame size in cells.
format	Explicit output format: markdown or html.

Value

A knitr::asis_output() object.

Label *Label widget*

Description

Displays text. See label().

Super class

Widget -> Label

Active bindings

text The label text (reactive).

Methods

Public methods:

- Label$new()
- Label$default_style()
- Label$update()
- Label$render()
- Label$clone()

Label$new(): Create a label.

Usage:

```
Label$new(text = "", id = NULL, classes = NULL, style = NULL, visible = TRUE)
```

Arguments:

text A string, character vector (one element per line) or span()s.

id, classes, style, visible See Widget.

Label$default_style(): The built-in style.

Usage:

```
Label$default_style()
```

Label$update(): Replace the text.

Usage:

```
Label$update(text = "")
```

Arguments:

text New text.

Label$render(): The label content.

Usage:

```
Label$render()
```

`Label$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`label` *Text label*

Description

Text label

Usage

```
label(text = "", id = NULL, classes = NULL, style = NULL, wrap = NULL)
```

Arguments

`text`	A string, a character vector (one element per line), a number, or `span()`s for styled text. A function of no arguments is re-evaluated whenever the `signal()`s it reads change.
`id`	Optional identifier (for `#id` selectors).
`classes`	Optional classes (for `.class` selectors).
`style`	A `style()`.
`wrap`	Shortcut for `style(wrap = )`: `"none"`, `"word"` or `"char"`.

Value

A `Label` widget. Change its text with `$update()` or by assigning `$text`.

Examples

```
lbl <- label("Hello", style = style(bold = TRUE))
lbl$update("Goodbye")
render_widget(lbl, 10, 1)
```

LogView *LogView widget*

Description

See log_view().

Super classes

Widget -> ListBase -> LogView

Public fields

has_cursor Log views scroll; they have no cursor.

max_lines Lines kept (older lines are dropped).

timestamps Prefix lines with the time?

auto_scroll Follow new lines while scrolled to the end?

Active bindings

lines The current lines.

paused Is following paused?

Methods

Public methods:

- LogView$new()
- LogView$write()
- LogView$clear()
- LogView$pause()
- LogView$resume()
- LogView$at_end()
- LogView$item_count()
- LogView$item_text()
- LogView$item_style()
- LogView$clone()

LogView$new(): Create a log view. See log_view().

Usage:

```
LogView$new(
  max_lines = 1000L,
  timestamps = FALSE,
  auto_scroll = TRUE,
  id = NULL,
```

```
  classes = NULL,
  style = NULL
)
```

Arguments:

`max_lines`, `timestamps`, `auto_scroll` See fields.

`id`, `classes`, `style` See Widget.

`LogView$write()`: Add lines.

Usage:

```
LogView$write(..., level = "info")
```

Arguments:

`...` Text, pasted together; newlines start new lines.

`level` `"debug"`, `"info"`, `"success"`, `"warning"` or `"error"`.

`LogView$clear()`: Remove all lines.

Usage:

```
LogView$clear()
```

`LogView$pause()`: Stop following new lines.

Usage:

```
LogView$pause()
```

`LogView$resume()`: Follow new lines again (and jump to the end).

Usage:

```
LogView$resume()
```

`LogView$at_end()`: Is the view scrolled to the last line?

Usage:

```
LogView$at_end()
```

`LogView$item_count()`: Number of lines.

Usage:

```
LogView$item_count()
```

`LogView$item_text()`: Text of line `i`.

Usage:

```
LogView$item_text(i)
```

Arguments:

`i` Index.

`LogView$item_style()`: Style of line `i` (by level).

Usage:

```
LogView$item_style(i)
```

Arguments:
`i` Index.

`LogView$clone()`: The objects of this class are cloneable with this method.

Usage:
```
LogView$clone(deep = FALSE)
```
Arguments:
`deep` Whether to make a deep clone.

`log_view` *Log view*

Description

A scrolling log for long-running work. `write()` appends lines with a level (`"debug"`, `"info"`, `"success"`, `"warning"`, `"error"`, shown in different colours). Only the last `max_lines` lines are kept, so memory stays bounded. While the view is scrolled to the end it follows new lines; scroll up (keys or wheel) to read older lines, or `pause()`.

Usage

```
log_view(
  max_lines = 1000L,
  timestamps = FALSE,
  auto_scroll = TRUE,
  id = NULL,
  classes = NULL,
  style = NULL
)
```

Arguments

`max_lines`	Number of lines kept.
`timestamps`	Prefix lines with `HH:MM:SS`?
`auto_scroll`	Follow new lines?
`id`	Optional identifier.
`classes`	Optional classes.
`style`	A `style()`.

Value

A `LogView` widget with methods `write()`, `clear()`, `pause()`, `resume()`.

Examples

```
logs <- log_view(max_lines = 500, id = "log")
logs$write("Starting model...")
logs$write("Convergence is slow", level = "warning")
```

MarkdownView *MarkdownView widget*

Description

A read-only Markdown document. See `markdown_view()`.

Super class

`Widget` -> `MarkdownView`

Active bindings

`markdown` The source text (assigning calls `set_markdown()`).

`links` The links in the document: a data frame (`text`, `url`).

Methods

Public methods:

- `MarkdownView$new()`
- `MarkdownView$default_style()`
- `MarkdownView$set_markdown()`
- `MarkdownView$render_lines()`
- `MarkdownView$content_width()`
- `MarkdownView$content_height()`
- `MarkdownView$clone()`

`MarkdownView$new()`: Create a Markdown view.

Usage:

```
MarkdownView$new(text = "", id = NULL, classes = NULL, style = NULL)
```

Arguments:

`text` Markdown source.

`id`, `classes`, `style` See Widget.

`MarkdownView$default_style()`: The built-in style.

Usage:

```
MarkdownView$default_style()
```

`MarkdownView$set_markdown()`: Replace the document.

Usage:

```
MarkdownView$set_markdown(text)
```

Arguments:

`text` Markdown source.

`MarkdownView$render_lines()`: The rendered lines for a width.

Usage:

```
MarkdownView$render_lines(width = NA_integer_)
```

Arguments:

`width` Content width.

`MarkdownView$content_width()`: Natural width.

Usage:

```
MarkdownView$content_width()
```

`MarkdownView$content_height()`: Natural height for a width.

Usage:

```
MarkdownView$content_height(width)
```

Arguments:

`width` Content width.

`MarkdownView$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`markdown_view` *Markdown view*

Description

Shows Markdown text: headings, paragraphs, **bold**, *italic*, ~~strikethrough~~, `inline code`, fenced code blocks, bullet and numbered lists, block quotes, horizontal rules, simple pipe tables and links. It is meant for help screens, release notes and dialogs; there is no HTML and the parser is not CommonMark-complete. Text is wrapped to the width of the view.

Usage

```
markdown_view(text, id = NULL, classes = NULL, style = NULL, scroll = TRUE)
```

Arguments

`text`	Markdown source (a character vector is joined with newlines).
`id`	Optional identifier.
`classes`	Optional classes.
`style`	A `style()`.
`scroll`	Wrap the view in a scrollable container?

Details

With `scroll = TRUE` (the default) the view is placed in a `scroll_view()` that fills its parent; use `scroll = FALSE` for short text inside your own layout.

Value

A `scroll_view()` containing a `MarkdownView` (or the `MarkdownView` itself with `scroll = FALSE`). Change the text with `$set_markdown()`; the `MarkdownView` is the first child of the scroll view.

Examples

```
md <- markdown_view("# Title\n\nSome *emphasis* and `code`.\n\n- one\n- two",
                    scroll = FALSE)
render_widget(md, 40, 8)$to_text()
```

Metric *Metric widget*

Description

See `metric()`.

Super class

`Widget` -> `Metric`

Public fields

`formatter function(value)` returning a string.

Active bindings

`value` The value (reactive).

`delta` The change shown next to the value (reactive).

Methods

Public methods:

- `Metric$new()`
- `Metric$default_style()`
- `Metric$update()`
- `Metric$render()`
- `Metric$clone()`

`Metric$new()`**:** Create a metric. See `metric()`.

Usage:

```
Metric$new(
  label,
  value = NA,
  delta = NULL,
  format = NULL,
  id = NULL,
  classes = NULL,
  style = NULL
)
```

Arguments:

`label` Label.

`value` Value.

`delta` Change.

`format` Formatter.

`id`, `classes`, `style` See Widget.

`Metric$default_style()`: The built-in style.

Usage:

```
Metric$default_style()
```

`Metric$update()`: Set the value (and optionally the delta).

Usage:

```
Metric$update(value, delta = NULL)
```

Arguments:

`value` New value.

`delta` New delta (`NULL` keeps none).

`Metric$render()`: Label, value and delta lines.

Usage:

```
Metric$render()
```

`Metric$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`metric` *Metric*

Description

A key number with a label and an optional change (delta) shown as a green up or red down arrow. Useful in dashboards.

Usage

```
metric(
  label,
  value = NA,
  delta = NULL,
  format = NULL,
  id = NULL,
  classes = NULL,
  style = NULL
)
```

Arguments

`label`	Label shown above the value.
`value`	The value (any R value; numbers are formatted with thousands separators).
`delta`	Optional change since the last value.
`format`	Optional `function(value)` returning a string (also used for the delta).
`id`	Optional identifier.
`classes`	Optional classes.
`style`	A `style()`.

Value

A `Metric` widget with reactive `value` and `delta`, and method `update(value, delta)`.

Examples

```
render_widget(metric("Accuracy", 0.943, delta = 0.012), 20, 4)
```

`modal` *Modal screens and dialogs*

Description

`modal()` builds a screen that is drawn over the current one and takes the keyboard and mouse until it is closed. Show it with `app$push_screen()`; close it with Escape or `screen$dismiss(result)`. The previously focused widget gets focus back.

Usage

```
modal(
  ...,
  title = NULL,
  width = 50L,
  dim = TRUE,
  dismissable = TRUE,
  on_dismiss = NULL,
  id = NULL
)

confirm_dialog(
  message,
  on_confirm = NULL,
  on_cancel = NULL,
  title = "Confirm",
  confirm_label = "OK",
  cancel_label = "Cancel",
  width = 50L
)

alert_dialog(message, title = "Message", button_label = "OK", width = 50L)
```

Arguments

`...`	Content widgets (placed in a bordered panel).
`title`	Panel title.
`width`	Panel width (cells, `"auto"`, `"50%"`, ...).
`dim`	Dim the screens below?
`dismissable`	Can Escape close the dialog?
`on_dismiss`	`function(result, app)` called when the dialog closes.
`id`	Optional identifier of the screen.
`message`	Message text.
`on_confirm, on_cancel`	`function(app)` called when the user confirms, or cancels (Cancel button or Escape).

`confirm_label`, `cancel_label`
: Button labels.

`button_label`
: Label of the close button.

Details

`confirm_dialog()` and `alert_dialog()` are ready-made dialogs built on `modal()`.

Value

A `ModalScreen`.

Examples

```
dlg <- confirm_dialog("Delete file?", on_confirm = function(app) message("deleted"))
a <- app(label("main"))
pilot <- test_app(a, 40, 10)
a$push_screen(dlg)
pilot$press("tab", "enter") # move to OK, press it
pilot$stop()
```

ModalScreen *Modal screen*

Description

A screen drawn over the current one. See `modal()`.

Super classes

`Widget` -> `Screen` -> `ModalScreen`

Public fields

`dim` Dim the screens below?

`dismissable` Close with Escape?

`on_dismiss` `NULL` or `function(result, app)` called when the screen is dismissed.

Methods

Public methods:

- `ModalScreen$new()`
- `ModalScreen$default_style()`
- `ModalScreen$default_bindings()`
- `ModalScreen$dismiss()`
- `ModalScreen$action_dismiss()`
- `ModalScreen$clone()`

`ModalScreen$new()`: Create a modal screen. See `modal()`.

Usage:

```
ModalScreen$new(
  ...,
  dim = TRUE,
  dismissable = TRUE,
  on_dismiss = NULL,
  id = NULL,
  classes = NULL,
  style = NULL
)
```

Arguments:

`...` Child widgets.

`dim`, `dismissable` See fields.

`on_dismiss` Callback.

`id`, `classes`, `style` See Widget.

`ModalScreen$default_style()`: The built-in style: content centred.

Usage:

```
ModalScreen$default_style()
```

`ModalScreen$default_bindings()`: Escape closes the screen (when dismissable).

Usage:

```
ModalScreen$default_bindings()
```

`ModalScreen$dismiss()`: Close the screen, passing `result` to the callbacks.

Usage:

```
ModalScreen$dismiss(result = NULL)
```

Arguments:

`result` Any value.

`ModalScreen$action_dismiss()`: Action used by the Escape binding.

Usage:

```
ModalScreen$action_dismiss()
```

`ModalScreen$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

motion_reduced *Reduced motion*

Description

Decorative animation (`animate()` transitions, indeterminate progress shimmer, spinners) is turned off when motion is reduced: `animate()` jumps to the final value at once, spinners show a still symbol. Timers, events and real progress updates are unaffected. Motion is reduced when `app(reduce_motion = TRUE)`, `options(termr.reduce_motion = TRUE)` or the environment variable `TERMR_REDUCE_MOTION=1` is set (in that order of precedence).

Usage

```
motion_reduced(app = NULL)
```

Arguments

app	An app, or `NULL` to look at the options only.

Value

`TRUE` or `FALSE`.

Examples

```
motion_reduced()
```

on *Event handlers*

Description

`on()` creates an event handler that can be passed to `app()` together with the widgets. It is equivalent to calling `app$on()`.

Usage

```
on(type, selector = NULL, handler)
```

Arguments

type	Event type such as `"button.pressed"`, `"input.submitted"`, `"key"`, `"resize"`, `"focus"` or `"mount"`. `"*"` matches any type.
selector	Optional selector (e.g. `"#hello"`, `".danger"`, `"Button"`) matched against the sender (for key and mouse events: the widget the event was sent to).
handler	`function(event, app)`.

Details

The handler runs when an event of the given type reaches the app and, if a selector is given, the widget that sent the event matches it.

Value

An object of class `termr_handler`.

Examples

```
on("button.pressed", "#hello", function(event, app) {
  app$query_one("#output")$update("Hello!")
})
```

`OptionList` *OptionList widget*

Description

A list of choices. See `option_list()`.

Super classes

`Widget` -> `ListBase` -> `OptionList`

Active bindings

`value` Value of the highlighted choice (`NULL` if none).

`choices` The choice values.

Methods

Public methods:

- `OptionList$new()`
- `OptionList$set_choices()`
- `OptionList$item_count()`
- `OptionList$item_text()`
- `OptionList$on_highlight()`
- `OptionList$on_activate()`
- `OptionList$clone()`

`OptionList$new()`**:** Create an option list. See `option_list()`.

Usage:

```
OptionList$new(
  choices,
  id = NULL,
  classes = NULL,
  style = NULL,
  disabled = FALSE
)
```

Arguments:

`choices` Choices.

`id`, `classes`, `style`, `disabled` See Widget.

`OptionList$set_choices()`: Replace the choices.

Usage:

```
OptionList$set_choices(choices)
```

Arguments:

`choices` Character vector (named: names are shown).

`OptionList$item_count()`: Number of choices.

Usage:

```
OptionList$item_count()
```

`OptionList$item_text()`: Label of choice `i`.

Usage:

```
OptionList$item_text(i)
```

Arguments:

`i` Index.

`OptionList$on_highlight()`: Sends `"option_list.highlighted"`.

Usage:

```
OptionList$on_highlight(i)
```

Arguments:

`i` Index.

`OptionList$on_activate()`: Sends `"option_list.selected"`.

Usage:

```
OptionList$on_activate(i)
```

Arguments:

`i` Index.

`OptionList$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

option_list *Option list*

Description

A scrollable list of choices. Up/Down/Page Up/Page Down/Home/End move the highlight (sending `"option_list.highlighted"`); Enter or a click selects (`"option_list.selected"`). `event$data` has `index`, `value` and `label`.

Usage

```
option_list(choices, id = NULL, classes = NULL, style = NULL)
```

Arguments

`choices`	A character vector. Names, if present, are shown instead of the values.
`id`	Optional identifier.
`classes`	Optional classes.
`style`	A `style()`.

Value

An `OptionList` widget with field `value`.

Examples

```
option_list(c(Comma = "csv", JSON = "json", Parquet = "parquet"))
```

Panel *Panel widget*

Description

A bordered container with an optional title. See `panel()`.

Super classes

`Widget` -> `Vertical` -> `Panel`

Methods

Public methods:

- `Panel$default_style()`
- `Panel$clone()`

`Panel$default_style()`: The built-in style.

Usage:

```
Panel$default_style()
```

`Panel$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`panel` *Panel*

Description

A container with a border and an optional title in the top border. Its height follows its content by default.

Usage

```
panel(..., title = NULL, id = NULL, classes = NULL, style = NULL)
```

Arguments

`...`	Child widgets.
`title`	Title shown in the top border.
`id`	Optional identifier.
`classes`	Optional classes.
`style`	A `style()`.

Value

A `Panel` widget. Change the title with `$border_title`.

Examples

```
render_widget(panel(label("CPU 42%"), title = "Metrics"), 20, 3)
```

patch_to_ansi *Convert a screen patch to ANSI escape sequences*

Description

Each run becomes a cursor movement followed by the run's characters. An SGR (style) sequence is emitted only when the style differs from the previous cell. The output always ends with a style reset.

Usage

```
patch_to_ansi(patch, color_mode = "truecolor")
```

Arguments

patch	A patch from diff_screen().
color_mode	One of "truecolor", "256", "16" or "none".

Value

A single string.

Examples

```
a <- screen_buffer(10, 1)
b <- a$copy()
b$put_text(1, 1, "hi", fg = "red")
patch_to_ansi(diff_screen(a, b))
```

ProcessView *ProcessView widget*

Description

Runs a program and shows its output. See process_view().

Super classes

Widget -> Vertical -> ProcessView

Public fields

command, args, wd, env, timeout The process to run.

autostart Start when the widget is mounted?

worker The Worker of the current run, or NULL.

Active bindings

`running` Is the process running?

`output` The output lines so far.

Methods

Public methods:

- `ProcessView$new()`
- `ProcessView$default_bindings()`
- `ProcessView$on_mount()`
- `ProcessView$start()`
- `ProcessView$cancel()`
- `ProcessView$restart()`
- `ProcessView$action_cancel_process()`
- `ProcessView$action_restart_process()`
- `ProcessView$clone()`

`ProcessView$new()`: Create a process view. See `process_view()`.

Usage:

```
ProcessView$new(
  command,
  args = character(),
  wd = NULL,
  env = NULL,
  timeout = NULL,
  autostart = TRUE,
  max_lines = 2000L,
  id = NULL,
  classes = NULL,
  style = NULL
)
```

Arguments:

`command`, `args`, `wd`, `env`, `timeout` The process to run.

`autostart` Start when mounted?

`max_lines` Output lines kept.

`id`, `classes`, `style` See Widget.

`ProcessView$default_bindings()`: Ctrl+X cancels the process, Ctrl+R restarts it.

Usage:

```
ProcessView$default_bindings()
```

`ProcessView$on_mount()`: Start the process when the widget is mounted.

Usage:

```
ProcessView$on_mount(event)
```

Arguments:

`event` A `MountEvent`.

`ProcessView$start()`: Run the process (output starts afresh).

Usage:

```
ProcessView$start()
```

`ProcessView$cancel()`: Stop the process.

Usage:

```
ProcessView$cancel()
```

`ProcessView$restart()`: Cancel the process and run it again.

Usage:

```
ProcessView$restart()
```

`ProcessView$action_cancel_process()`: Cancel action.

Usage:

```
ProcessView$action_cancel_process()
```

`ProcessView$action_restart_process()`: Restart action.

Usage:

```
ProcessView$action_restart_process()
```

`ProcessView$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`process_view` *Process view*

Description

Runs an external program (without a shell) and shows its output in a scrolling log: standard output as normal lines and standard error in the error colour, under a one-line status. The process starts when the view is mounted (or call `$start()`), is cancelled when the view is removed or the app exits, and sends `"process.finished"` (`status`, `state`) when it ends. Ctrl+X cancels it and Ctrl+R restarts it while the view has focus. **Experimental.**

Usage

```
process_view(
  command,
  args = character(),
  wd = NULL,
  env = NULL,
  timeout = NULL,
  autostart = TRUE,
  max_lines = 2000L,
  id = NULL,
  classes = NULL,
  style = NULL
)
```

Arguments

command	The program (a path or a name on the PATH).
args	Arguments, one element each; never interpreted by a shell.
wd	Working directory (NULL: current).
env	Named character vector of extra environment variables.
timeout	Seconds before the process is stopped (NULL: no limit).
autostart	Start when mounted?
max_lines	Output lines kept.
id, classes, style	Common widget arguments.

Value

A ProcessView widget with $start(), $cancel(), $restart(), $running, $output and $worker.

Examples

```
pv <- process_view(file.path(R.home("bin"), "Rscript"), c("-e", "cat('hi\\n')"),
                   autostart = FALSE)
pv$running
```

ProgressBar *ProgressBar widget*

Description

See progress_bar().

Super class

Widget -> ProgressBar

Public fields

`paint_states` Progress changes only repaint.

`total` The value that means 100%.

`show_percent` Show the percentage?

Active bindings

`value` Progress between 0 and `total`, or `NA` (reactive).

`fraction` Progress as a fraction (`NA` when indeterminate).

Methods

Public methods:

- `ProgressBar$new()`
- `ProgressBar$default_style()`
- `ProgressBar$on_mount()`
- `ProgressBar$advance()`
- `ProgressBar$render_lines()`
- `ProgressBar$content_width()`
- `ProgressBar$clone()`

`ProgressBar$new()`: Create a progress bar. See `progress_bar()`.

Usage:

```
ProgressBar$new(
  value = 0,
  total = 1,
  show_percent = TRUE,
  id = NULL,
  classes = NULL,
  style = NULL
)
```

Arguments:

`value` Progress (or `NA` for indeterminate).

`total` Total.

`show_percent` Show the percentage?

`id`, `classes`, `style` See Widget.

`ProgressBar$default_style()`: The built-in style.

Usage:

```
ProgressBar$default_style()
```

`ProgressBar$on_mount()`: Animate the indeterminate state.

Usage:

```
ProgressBar$on_mount(event)
```

Arguments:

`event` A `MountEvent`.

`ProgressBar$advance()`: Advance the progress.

Usage:

```
ProgressBar$advance(amount = 1)
```

Arguments:

`amount` Amount to add.

`ProgressBar$render_lines()`: The bar for a given width.

Usage:

```
ProgressBar$render_lines(width = NA_integer_)
```

Arguments:

`width` Content width.

`ProgressBar$content_width()`: Natural width.

Usage:

```
ProgressBar$content_width()
```

`ProgressBar$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`progress_bar` *Progress bar*

Description

A bar filling as `value` goes from 0 to `total`, with a percentage. Set `value = NA` for an indeterminate (animated) bar.

Usage

```
progress_bar(
  value = 0,
  total = 1,
  show_percent = TRUE,
  id = NULL,
  classes = NULL,
  style = NULL
)
```

Arguments

value	Current progress, or NA.
total	Value that means complete.
show_percent	Show the percentage?
id	Optional identifier.
classes	Optional classes.
style	A style().

Value

A ProgressBar with a reactive value and method advance().

Examples

```
render_widget(progress_bar(0.42), 20, 1)
```

PropertyGrid *PropertyGrid widget*

Description

See property_grid().

Super classes

Widget -> ScrollView -> PropertyGrid

Public fields

max_key_width Maximum key width in terminal cells.

formatter Optional function(name, value).

Active bindings

data The current named data (reactive).

Methods

Public methods:

- PropertyGrid$new()
- PropertyGrid$default_style()
- PropertyGrid$set_data()
- PropertyGrid$clone()

PropertyGrid$new(): Create a property grid. See property_grid().

Usage:

```
PropertyGrid$new(
  data = list(),
  formatter = NULL,
  max_key_width = 20L,
  id = NULL,
  classes = NULL,
  style = NULL,
  disabled = FALSE
)
```

Arguments:

`data` Named list/vector or data frame.

`formatter` Optional value formatter.

`max_key_width` Maximum key width.

`id`, `classes`, `style`, `disabled` See `property_grid()`.

`PropertyGrid$default_style()`: The built-in scrollable style.

Usage:

```
PropertyGrid$default_style()
```

`PropertyGrid$set_data()`: Replace all rows.

Usage:

```
PropertyGrid$set_data(data)
```

Arguments:

`data` New named data.

`PropertyGrid$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`property_grid` *Key/value property inspector*

Description

Displays named list/vector entries in two aligned columns. Keys are sized to their longest name up to `max_key_width`; values wrap and the view scrolls when needed. `NULL`, `NA`, dates and vectors have compact default formatting. Lists and other objects are shown as placeholders instead of being recursively printed. A custom `formatter(name, value)` can replace the default value formatting. `data` may be a reactive function of no arguments.

Usage

```
property_grid(
  data = list(),
  format = NULL,
  max_key_width = 20L,
  id = NULL,
  classes = NULL,
  style = NULL,
  disabled = FALSE
)
```

Arguments

`data`	Named list or vector; data frames are accepted as named columns. `NULL` displays an empty grid.
`format`	Optional `function(name, value)` returning display text.
`max_key_width`	Maximum key column width in terminal cells.
`id`	Optional identifier.
`classes`	Optional classes.
`style`	A `style()`.
`disabled`	Disable focus and scrolling interaction?

Value

A `PropertyGrid` (a focusable, scrollable widget) with reactive `$data` and `$set_data()`.

Choosing a key/value widget

- `key_value()`: a few known values shown in full, such as a summary panel. Not focusable, does not scroll; values are not wrapped.
- `property_grid()`: an inspector for many or arbitrary named values. Focusable and scrollable, wraps long values, formats vectors and lists compactly and accepts a `format` function.
- `record_view()`: `property_grid()` for exactly one record, such as the selected row of a `data_table()`; accepts a one-row data frame.

Examples

```
property_grid(list(type = "numeric", missing = 12, unique = 345))
```

RadioButton *RadioButton widget*

Description

One option of a radio_set().

Super class

Widget -> RadioButton

Public fields

focusable Radio buttons can be focused.

value The value this option stands for.

Active bindings

selected Is this option selected? (read-only; use the set)

label The label (reactive).

Methods

Public methods:

- RadioButton$new()
- RadioButton$default_style()
- RadioButton$default_bindings()
- RadioButton$action_choose()
- RadioButton$on_click()
- RadioButton$render()
- RadioButton$clone()

RadioButton$new(): Create a radio button. See radio_button().

Usage:

```
RadioButton$new(
  label,
  value = label,
  id = NULL,
  classes = NULL,
  style = NULL,
  disabled = FALSE
)
```

Arguments:

label Label.

`value` Value.
`id`, `classes`, `style`, `disabled` See Widget.

`RadioButton$default_style()`: The built-in style.

Usage:

```
RadioButton$default_style()
```

`RadioButton$default_bindings()`: Space and Enter select the option.

Usage:

```
RadioButton$default_bindings()
```

`RadioButton$action_choose()`: Select this option in its radio set.

Usage:

```
RadioButton$action_choose()
```

`RadioButton$on_click()`: Clicking selects.

Usage:

```
RadioButton$on_click(event)
```

Arguments:

`event` A `MouseEvent`.

`RadioButton$render()`: The dot and the label.

Usage:

```
RadioButton$render()
```

`RadioButton$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`RadioSet` *RadioSet widget*

Description

A group of radio buttons. See `radio_set()`.

Super classes

`Widget` -> `Vertical` -> `RadioSet`

Active bindings

`value` The selected value (`NULL` if none); assign to select.

Methods

Public methods:

- RadioSet$new()
- RadioSet$default_style()
- RadioSet$default_bindings()
- RadioSet$select()
- RadioSet$action_next_option()
- RadioSet$action_previous_option()
- RadioSet$clone()

`RadioSet$new()`: Create a radio set. See `radio_set()`.

Usage:

```
RadioSet$new(
  ...,
  selected = NULL,
  id = NULL,
  classes = NULL,
  style = NULL,
  disabled = FALSE
)
```

Arguments:

`...` Radio buttons.

`selected` Value to select initially.

`id`, `classes`, `style`, `disabled` See Widget.

`RadioSet$default_style()`: The built-in style.

Usage:

```
RadioSet$default_style()
```

`RadioSet$default_bindings()`: Up/Down move to the previous / next option and select it.

Usage:

```
RadioSet$default_bindings()
```

`RadioSet$select()`: Select an option (sends `"radio_set.changed"`).

Usage:

```
RadioSet$select(button)
```

Arguments:

`button` A child radio button.

`RadioSet$action_next_option()`: Move to the next option.

Usage:

```
RadioSet$action_next_option()
```

`RadioSet$action_previous_option()`: Move to the previous option.

Usage:

```
RadioSet$action_previous_option()
```

`RadioSet$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`radio_set` *Radio buttons*

Description

`radio_set()` groups `radio_button()`s so that one can be selected. Space, Enter or a click selects an option; Up/Down move between options. Changes send `"radio_set.changed"` (`event$data`: `value`, `index`, `label`) from the set.

Usage

```
radio_set(..., selected = NULL, id = NULL, classes = NULL, style = NULL)

radio_button(label, value = label, id = NULL, disabled = FALSE)
```

Arguments

`...`	`radio_button()`s.
`selected`	Value selected initially.
`id`	Optional identifier.
`classes`	Optional classes.
`style`	A `style()`.
`label`	Label of the option.
`value`	Value of the option (defaults to the label).
`disabled`	Can the option be selected?

Value

A `RadioSet` widget with a `value` field.

Examples

```
radio_set(radio_button("CSV", "csv"), radio_button("JSON", "json"), selected = "csv")
```

reactive *Reactive state fields*

Description

`reactive()` declares a state field of a custom widget (see `widget()`). Reading the field returns its value; assigning a different value (`self$count <- self$count + 1L`) invalidates the widget and schedules a repaint. Repaints are batched: any number of assignments in one event-loop tick produce a single render and a single screen diff.

Usage

```
reactive(value = NULL, watch = NULL)
```

Arguments

`value`	Initial value.
`watch`	Optional `function(self, value, old)` called after the value changed, e.g. to post a message or update other widgets.

Value

An object of class `termr_reactive`.

Examples

```
reactive(0L)
reactive("", watch = function(self, value, old) message("now ", value))
```

record_view *View one record*

Description

`record_view()` accepts a named list/vector or exactly one data-frame row and presents it through the same renderer as `property_grid()`. It is a semantic convenience wrapper, not a separate rendering system. A reactive function may return a new record whenever its signals change.

Usage

```
record_view(
  data,
  format = NULL,
  max_key_width = 20L,
  id = NULL,
  classes = NULL,
  style = NULL,
  disabled = FALSE
)
```

Arguments

data	One named list, named vector, or one-row data frame. NULL displays an empty view.
format	Optional function(name, value) formatter.
max_key_width	Maximum key column width.
id	Optional identifier.
classes	Optional classes.
style	A style().
disabled	Disable focus and scrolling interaction?

Value

A PropertyGrid widget.

Choosing a key/value widget

- key_value(): a few known values shown in full, such as a summary panel. Not focusable, does not scroll; values are not wrapped.
- property_grid(): an inspector for many or arbitrary named values. Focusable and scrollable, wraps long values, formats vectors and lists compactly and accepts a format function.
- record_view(): property_grid() for exactly one record, such as the selected row of a data_table(); accepts a one-row data frame.

Examples

```
record_view(list(id = 42, name = "Alice", active = TRUE))
```

region *Screen regions*

Description

A region (rectangle) describes a region of the terminal in cells. Coordinates are 1-based, like R indices and ANSI cursor positions: `x` is the column and `y` is the row of the top-left cell.

Usage

```
region(x = 1L, y = 1L, width = 0L, height = 0L)
```

Arguments

`x`, `y`	Column and row of the top-left cell.
`width`, `height`	Size in cells. Negative sizes are clamped to zero.

Details

Rectangles are plain values (lists with class `termr_rect`), so they can be compared with `identical()` and are cheap to create during layout.

Value

An object of class `termr_rect`.

Examples

```
r <- region(1, 1, 40, 5)
r
```

register_layout *Register an experimental custom container layout*

Description

A registered name can be used as `layout` in `style()`. Custom names start with `custom_` to keep built-in names reserved.

Usage

```
register_layout(name, arrange, measure, replace = FALSE)
```

Arguments

name	A unique custom_ name.
arrange	function(children, inner, parent_style) returning one region() per child.
measure	function(children, parent_style) returning width() and height(width) functions.
replace	Replace an existing registration of name? Packages that register in .onLoad() can pass TRUE so that reloading them works. Built-in layouts can never be replaced (their names do not start with custom_).

Value

The registered name, invisibly.

---

render_html *Render a styled HTML fragment*

Description

Render a styled HTML fragment

Usage

```
render_html(x, width = 80, height = 24, color = TRUE, trim = TRUE)
```

Arguments

x	A widget, Screen, App, or ScreenBuffer.
width, height	Frame size in cells.
color	Include framebuffer colours even when NO_COLOR is set?
trim	Remove trailing spaces from each line?

Value

An embeddable <pre> fragment. Styling is approximate and depends on the browser's monospace font and CSS environment.

Examples

```
cat(render_html(label("Ready", style = style(foreground = "green", bold = TRUE))))
```

render_markdown *Render a terminal frame as a Markdown code block*

Description

Render a terminal frame as a Markdown code block

Usage

```
render_markdown(x, width = 80, height = 24, trim = TRUE)
```

Arguments

x	A widget, Screen, App, or ScreenBuffer.
width, height	Frame size in cells.
trim	Remove trailing spaces from each line?

Value

A fenced Markdown text block.

Examples

```
cat(render_markdown(label("Hello"), width = 20, height = 2))
```

render_svg *Render a static frame as SVG*

Description

Render a static frame as SVG

Usage

```
render_svg(
  x,
  width = 80,
  height = 24,
  cell_width = 9,
  line_height = 18,
  font_family = "monospace",
  color = TRUE
)
```

Arguments

x	A widget, Screen, App, or ScreenBuffer.
width, height	Frame size in cells.
cell_width	Width of a terminal cell in SVG user units.
line_height	Height of a row in SVG user units.
font_family	CSS font family.
color	Include framebuffer colours?

Value

An SVG document string.

---

render_text *Render a widget tree as plain text*

Description

Static renderers use the normal layout and paint pipeline and do not start an event loop or write to a terminal.

Usage

```
render_text(x, width = 80, height = 24, trim = TRUE)

render_lines(x, width = 80, height = 24, trim = TRUE)
```

Arguments

x	A widget, Screen, App, or ScreenBuffer.
width, height	Frame size in cells.
trim	Remove trailing spaces from each line?

Value

`render_text()` returns one string separated by newlines; `render_lines()` returns one string per row.

Examples

```
render_text(panel(label("Finished"), title = "Job"), width = 20, height = 5)
```

render_widget *Render widgets to a virtual screen*

Description

Lays out and paints a widget tree into a ScreenBuffer without a terminal. Useful for snapshot tests and for checking layouts interactively.

Usage

```
render_widget(x, width = 80L, height = 24L)
```

Arguments

x	A widget, a Screen or an App.
width, height	Screen size in cells.

Value

A ScreenBuffer; use `$to_text()` for plain text.

Examples

```
buf <- render_widget(vertical(label("a"), label("b")), 10, 2)
buf$to_text()
```

Rule *Rule widget*

Description

See `rule()`.

Super class

`Widget` -> `Rule`

Public fields

`orientation` `"horizontal"` or `"vertical"`.

`title` Optional title in the middle of a horizontal rule.

Methods

Public methods:

- Rule$new()
- Rule$default_style()
- Rule$paint()
- Rule$clone()

`Rule$new()`: Create a rule. See `rule()`.

Usage:

```
Rule$new(
  orientation = "horizontal",
  title = NULL,
  id = NULL,
  classes = NULL,
  style = NULL
)
```

Arguments:

`orientation` Orientation.

`title` Title.

`id`, `classes`, `style` See Widget.

`Rule$default_style()`: The built-in style.

Usage:

```
Rule$default_style()
```

`Rule$paint()`: Paint the line.

Usage:

```
Rule$paint(buffer, area, st)
```

Arguments:

`buffer` A ScreenBuffer.

`area` Visible part of the region.

`st` Computed style.

`Rule$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

rule *Rule (separator)*

Description

A horizontal or vertical line, optionally with a title.

Usage

```
rule(
  orientation = "horizontal",
  title = NULL,
  id = NULL,
  classes = NULL,
  style = NULL
)
```

Arguments

orientation	"horizontal" or "vertical".
title	Optional title for horizontal rules.
id	Optional identifier.
classes	Optional classes.
style	A style().

Value

A Rule widget.

Examples

```
render_widget(rule(title = "Results"), 20, 1)
```

run *Run an application*

Description

Takes over the terminal (alternate screen, raw keyboard input), runs the event loop until the app calls exit(), and restores the terminal - also when an error occurs or the user interrupts R.

Usage

```
run(x, ...)
```

Arguments

x	An App created by app(), or a widget (wrapped in an app).
...	Passed to App$run(), e.g. driver.

Details

When the app stops, its timers, widget timers, workers and animations are cancelled. The same app can be run again, but timers and workers have to be started again (for example from a "mount" handler).

termr apps need a real terminal: run them with Rscript (or an interactive R session) in a terminal such as Windows Terminal, iTerm2 or GNOME Terminal. RStudio's console is not a terminal.

Value

The value passed to app$exit(), invisibly.

---

run_example *Run a bundled example app*

Description

Run a bundled example app

Usage

```
run_example(name = NULL)
```

Arguments

name	Example name. Call run_example() without arguments to list the available examples.

Value

The example names (invisibly when an example was run).

Examples

```
run_example()
```

run_worker *Background workers and subprocesses*

Description

app$run_worker() runs a function in a separate R process and reports back through events and callbacks, so the interface stays responsive during long computations (model fitting, downloads, file processing). app$run_process() does the same for an external program.

Usage

```
run_worker(
  fn,
  args = list(),
  on_complete = NULL,
  on_error = NULL,
  on_progress = NULL,
  name = NULL,
  owner = NULL,
  packages = character(),
  inline = FALSE,
  on_stdout = NULL,
  on_stderr = NULL,
  timeout = NULL,
  app = current_app()
)

run_process(
  command,
  args = character(),
  on_complete = NULL,
  on_error = NULL,
  on_stdout = NULL,
  on_stderr = NULL,
  on_exit = NULL,
  timeout = NULL,
  wd = NULL,
  env = NULL,
  name = NULL,
  owner = NULL,
  app = current_app()
)
```

Arguments

fn	The function to run.
args	A list of arguments.

on_complete	function(result, app).
on_error	function(message, app).
on_progress	function(value, message, app).
name	Name of the worker.
owner	A widget owning the worker (events come from it, and it is cancelled when the widget is removed).
packages	Packages to attach in the worker process.
inline	Run in this process instead (for tests).
on_stdout, on_stderr	function(line, app) called for each output line.
timeout	Seconds after which the worker is stopped and fails (NULL: no limit).
app	The app (defaults to the running app).
command	The program to run (a path or a name found on the PATH).
on_exit	function(status, app) called when the process ends (whatever the status).
wd	Working directory (NULL: the current one).
env	Named character vector of extra environment variables.

Details

The function runs in a fresh R session (Rscript --vanilla), so it cannot directly access the caller's global environment. Lexical values referenced by the function are captured by value in a small environment; unrelated bindings in the enclosing environment are omitted. For example, a function using a local scalar binding works without serializing other locals. Referenced nested and recursive helper functions are captured too. Dynamic lexical lookup (get(), eval() or parent.frame()) and active bindings are rejected; pass those values explicitly through args. Prefer args for large data and list required packages in packages. Inside the function, termr_progress(value, message) reports progress on a channel of its own, so anything the function prints to stdout or stderr is free for on_stdout / on_stderr (message() goes to stderr). The child receives the parent's library search paths, and R_TESTS is cleared so a check-session startup hook cannot rerun the test suite.

run_process(command, args) starts a program *without a shell*: the command and its arguments are separate, so nothing is interpolated or interpreted. (To use a shell on purpose, run it yourself: run_process("sh", c("-c", script)).) It completes with the exit status as result; a non-zero status counts as failure.

Events (from the owner widget, or the screen), with event$data$worker: "worker.started", "worker.progress" (value, message), "worker.stdout" / "worker.stderr" (line), "worker.completed" (result), "worker.failed" (error, timed_out, status), "worker.cancelled". Output lines are stripped of control characters; at most 200 lines are delivered per event-loop tick.

Workers stop when cancel() is called, when their owner widget is removed, when timeout seconds have passed, and when the app exits. A stopping worker is interrupted first (Unix) and killed, with its child processes, if it does not end within a short grace period.

Temporary files (the job, the result and the progress file) have unique names and are removed when the worker ends, however it ends.

The job (the function with its referenced lexical bindings, and args) is serialized to a temporary file. Explicitly referenced objects and values in args are still serialized. Workers and programs

inherit the app's environment variables: avoid putting secrets into worker closures or arguments unnecessarily.

There is no worker pool: start as many workers as you need; each is an independent process. This API is **experimental**.

Value

A Worker handle with `cancel()`, `state`, `result`, `error`, `stdout` and `stderr`.

Testing

`inline = TRUE` runs the function in the current R process at once (blocking), which makes tests fast and deterministic. With real processes, `pilot$wait_for_workers()` waits until they finish.

Examples

```
pilot <- test_app(app(label("Jobs")), width = 30, height = 1)
# `inline = TRUE` runs the job in this process; without it, the same call
# starts a background R process, where `termr_progress()` is available.
pilot$app$run_worker(
  function(n) sum(seq_len(n)),
  args = list(n = 20),
  inline = TRUE,
  on_complete = function(result, app) message("result: ", result)
)
pilot$step()
pilot$stop()
```

---

r_highlighter *Lightweight R syntax highlighter*

Description

Returns a viewport-first tokenizer for `text_area()`. It highlights common R keywords, strings, comments, numeric constants, operators, and function names. It uses tolerant regular expressions and accepts incomplete syntax. Each line produces grapheme-indexed spans with `start`, `end`, and `token`.

Usage

```
r_highlighter()
```

Value

A function `(lines, state = NULL)` suitable for `text_area()`.

Screen *Screen widget*

Description

The root of a widget tree. An App shows one screen at a time; the screen always fills the terminal. Bindings on the screen apply to every widget in it.

Super class

`Widget` -> `Screen`

Methods

Public methods:

- `Screen$default_style()`
- `Screen$clone()`

`Screen$default_style()`: The built-in style.

Usage:

```
Screen$default_style()
```

`Screen$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

ScreenBuffer *Virtual screen (framebuffer)*

Description

A `ScreenBuffer` is an in-memory grid of terminal cells. Widgets are painted into a buffer, buffers are compared with `diff_screen()` and only the differences are sent to the terminal.

Cells are stored in four `height x width` matrices (`chars`, `fg`, `bg`, `attrs`) indexed as `[row, column]`. A wide character occupies two columns: the first holds the character, the second holds `""` (a continuation cell). All writes keep this invariant, so a wide character is never left half-overwritten.

Use `screen_buffer()` to create one.

Public fields

`width` Number of columns.

`height` Number of rows.

`chars` Character matrix of cell contents.

`fg` Character matrix of canonical foreground colours.

`bg` Character matrix of canonical background colours.

`attrs` Integer matrix of attribute bit masks.

Methods

Public methods:

- `ScreenBuffer$new()`
- `ScreenBuffer$clear()`
- `ScreenBuffer$bounds()`
- `ScreenBuffer$get_cell()`
- `ScreenBuffer$set_cell()`
- `ScreenBuffer$fill()`
- `ScreenBuffer$paint_background()`
- `ScreenBuffer$put_text()`
- `ScreenBuffer$resize()`
- `ScreenBuffer$copy()`
- `ScreenBuffer$equals()`
- `ScreenBuffer$to_text()`
- `ScreenBuffer$print()`
- `ScreenBuffer$clone()`

`ScreenBuffer$new()`: Create a blank buffer.

Usage:

```
ScreenBuffer$new(width, height)
```

Arguments:

`width`, `height` Size in cells.

`ScreenBuffer$clear()`: Reset every cell to a blank default cell.

Usage:

```
ScreenBuffer$clear()
```

`ScreenBuffer$bounds()`: The full-buffer rectangle.

Usage:

```
ScreenBuffer$bounds()
```

`ScreenBuffer$get_cell()`: Read one cell.

Usage:

```
ScreenBuffer$get_cell(x, y)
```

Arguments:

`x, y` Column and row.

`ScreenBuffer$set_cell()`: Write one cell (see `cell()`).

Usage:

```
ScreenBuffer$set_cell(x, y, cell)
```

Arguments:

`x, y` Column and row.
`cell` A `termr_cell`.

`ScreenBuffer$fill()`: Fill a region with one character and style.

Usage:

```
ScreenBuffer$fill(region = NULL, char = " ", fg = NULL, bg = NULL, attrs = 0L)
```

Arguments:

`region` A `region()`; defaults to the whole buffer.
`char` A single-width character.
`fg, bg` Colours (see `style()`).
`attrs` Attribute bit mask.

`ScreenBuffer$paint_background()`: Set only the background colour of a region.

Usage:

```
ScreenBuffer$paint_background(region, bg)
```

Arguments:

`region` A `region()`.
`bg` Colour.

`ScreenBuffer$put_text()`: Write a single line of text.
Text is clipped to `clip` (and to the buffer). A wide character that would straddle the clip edge is replaced by a space. Control characters are removed.

Usage:

```
ScreenBuffer$put_text(
  x,
  y,
  text,
  fg = NULL,
  bg = NULL,
  attrs = 0L,
  clip = NULL
)
```

Arguments:

`x, y` Column and row of the first character.
`text` A single string (no newlines).

`fg` Foreground colour.

`bg` Background colour. `NULL` keeps the existing background, so text can be drawn over a painted widget background.

`attrs` Attribute bit mask.

`clip` Optional clipping `region()`.

Returns: The column after the last character, invisibly.

`ScreenBuffer$resize()`**:** Change the buffer size, keeping the top-left content.

Usage:

```
ScreenBuffer$resize(width, height)
```

Arguments:

`width, height` New size.

`ScreenBuffer$copy()`**:** An independent copy of the buffer.

Usage:

```
ScreenBuffer$copy()
```

`ScreenBuffer$equals()`**:** Do two buffers hold exactly the same cells?

Usage:

```
ScreenBuffer$equals(other)
```

Arguments:

`other` Another `ScreenBuffer`.

`ScreenBuffer$to_text()`**:** The buffer as plain text, one string per row.

Usage:

```
ScreenBuffer$to_text()
```

`ScreenBuffer$print()`**:** Print the buffer contents framed by a border.

Usage:

```
ScreenBuffer$print(...)
```

Arguments:

`...` Ignored.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

screen_buffer *Create a virtual screen buffer*

Description

Create a virtual screen buffer

Usage

```
screen_buffer(width, height)
```

Arguments

width, height	Size in cells.

Value

A ScreenBuffer.

Examples

```
buf <- screen_buffer(10, 2)
buf$put_text(2, 1, "hello", fg = "red")
buf$to_text()
```

screen_snapshot *Create a deterministic screen snapshot*

Description

The returned base R list is independent of jsonlite and has a stable schema_version field. Runs are row-major and represent adjacent cells with the same foreground, background, and attribute mask.

Usage

```
screen_snapshot(x, width = 80, height = 24)
```

Arguments

x	A widget, Screen, App, or ScreenBuffer.
width, height	Frame size in cells.

Value

A termr_screen_snapshot list.

Examples

```
screen_snapshot(label("Hello"), width = 20, height = 2)
```

---

screen_snapshot_json *Serialize a screen snapshot as JSON*

Description

The JSON form of `screen_snapshot()`. Requires the optional `jsonlite` package.

Usage

```
screen_snapshot_json(x, width = 80, height = 24, pretty = TRUE)
```

Arguments

`x`	A widget, Screen, App, or ScreenBuffer.
`width, height`	Frame size in cells.
`pretty`	Format the JSON for readability?

Value

A JSON string.

ScrollView *ScrollView widget*

Description

A container whose content can be larger than the widget. See `scroll_view()`.

Super class

`Widget` -> `ScrollView`

Public fields

`focusable` Scroll views can take focus so they can be scrolled with the keyboard even without focusable children.

`direction` `"vertical"`, `"horizontal"` or `"both"`.

`scrollbars` Draw scroll bars when the content overflows?

Active bindings

`offset_x`, `offset_y` Scroll offsets in cells (reactive).

`viewport` The visible content rectangle (after the last layout).

`virtual_size` Size of the scrollable content: `c(width, height)`.

`max_scroll` Largest offsets: `c(x, y)`.

Methods

Public methods:

- `ScrollView$new()`
- `ScrollView$default_style()`
- `ScrollView$default_bindings()`
- `ScrollView$scroll_to()`
- `ScrollView$scroll_by()`
- `ScrollView$scroll_home()`
- `ScrollView$scroll_end()`
- `ScrollView$scroll_into_view()`
- `ScrollView$on_mouse_scroll()`
- `ScrollView$action_scroll_up()`
- `ScrollView$action_scroll_down()`
- `ScrollView$action_scroll_left()`
- `ScrollView$action_scroll_right()`
- `ScrollView$action_page_up()`
- `ScrollView$action_page_down()`
- `ScrollView$action_scroll_home()`
- `ScrollView$action_scroll_end()`
- `ScrollView$arrange_children()`
- `ScrollView$layout_descend()`
- `ScrollView$child_clip()`
- `ScrollView$paint()`
- `ScrollView$clone()`

`ScrollView$new()`**:** Create a scroll view. See `scroll_view()`.

Usage:

```
ScrollView$new(
  ...,
  direction = "vertical",
  scrollbars = TRUE,
  focusable = TRUE,
  id = NULL,
  classes = NULL,
  style = NULL,
  disabled = FALSE
)
```

Arguments:

`...` Child widgets.

`direction` Scroll direction.

`scrollbars` Show scroll bars?

`focusable` Can the scroll view itself be focused?

`id`, `classes`, `style`, `disabled` See Widget.

`ScrollView$default_style()`: The built-in style.

Usage:

```
ScrollView$default_style()
```

`ScrollView$default_bindings()`: Arrow keys, Page Up/Down and Home/End scroll.

Usage:

```
ScrollView$default_bindings()
```

`ScrollView$scroll_to()`: Scroll to an absolute position (in cells). Positions are clamped to the scrollable range at the next layout.

Usage:

```
ScrollView$scroll_to(x = NULL, y = NULL)
```

Arguments:

`x`, `y` New offsets; `NULL` keeps the current one.

`ScrollView$scroll_by()`: Scroll relative to the current position.

Usage:

```
ScrollView$scroll_by(dx = 0L, dy = 0L)
```

Arguments:

`dx`, `dy` Cells to scroll (negative: up/left).

`ScrollView$scroll_home()`: Scroll to the top (and left).

Usage:

```
ScrollView$scroll_home()
```

`ScrollView$scroll_end()`: Scroll to the bottom.

Usage:

```
ScrollView$scroll_end()
```

`ScrollView$scroll_into_view()`: Scroll so that a descendant is fully visible (as far as possible). Uses the last layout; the app re-lays out afterwards.

Usage:

```
ScrollView$scroll_into_view(widget)
```

Arguments:

`widget` A descendant widget.

Returns: `TRUE` if the offsets changed.

`ScrollView$on_mouse_scroll()`: The mouse wheel scrolls by three lines. When the view cannot scroll further, the event bubbles to outer scroll views.

Usage:

```
ScrollView$on_mouse_scroll(event)
```

Arguments:

`event` A `MouseEvent`.

`ScrollView$action_scroll_up()`: Scroll one line up.

Usage:

```
ScrollView$action_scroll_up()
```

`ScrollView$action_scroll_down()`: Scroll one line down.

Usage:

```
ScrollView$action_scroll_down()
```

`ScrollView$action_scroll_left()`: Scroll one column left.

Usage:

```
ScrollView$action_scroll_left()
```

`ScrollView$action_scroll_right()`: Scroll one column right.

Usage:

```
ScrollView$action_scroll_right()
```

`ScrollView$action_page_up()`: Scroll one page up.

Usage:

```
ScrollView$action_page_up()
```

`ScrollView$action_page_down()`: Scroll one page down.

Usage:

```
ScrollView$action_page_down()
```

`ScrollView$action_scroll_home()`: Scroll to the top.

Usage:

```
ScrollView$action_scroll_home()
```

`ScrollView$action_scroll_end()`: Scroll to the bottom.

Usage:

```
ScrollView$action_scroll_end()
```

`ScrollView$arrange_children()`: Lay the children out in a virtual area, shifted by the scroll offsets; reserves space for scroll bars when needed.

Usage:

```
ScrollView$arrange_children(children, inner, st)
```

Arguments:

`children` Visible children.

`inner` Content rectangle.

`st` Computed style.

`ScrollView$layout_descend()`: Children entirely outside the viewport are not laid out in detail (unless a complete layout is needed, e.g. to scroll a focused widget into view).

Usage:

```
ScrollView$layout_descend(rects)
```

Arguments:

`rects` Child rectangles.

`ScrollView$child_clip()`: Children are clipped to the viewport.

Usage:

```
ScrollView$child_clip(st)
```

Arguments:

`st` Computed style.

`ScrollView$paint()`: Paint background, border and scroll bars.

Usage:

```
ScrollView$paint(buffer, area, st)
```

Arguments:

`buffer` A ScreenBuffer.

`area` Visible part of the region.

`st` Computed style.

`ScrollView$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`scroll_view` *Scrollable container*

Description

Shows content that is larger than the available space. The content is laid out at its natural size and the visible part (the viewport) is shifted by the scroll offsets; everything outside is clipped. Scroll bars appear when the content overflows.

Usage

```
scroll_view(
  ...,
  direction = "vertical",
  scrollbars = TRUE,
  focusable = TRUE,
  id = NULL,
  classes = NULL,
  style = NULL
)
```

Arguments

`...`	Child widgets.
`direction`	`"vertical"` (default), `"horizontal"` or `"both"`.
`scrollbars`	Draw scroll bars when the content overflows?
`focusable`	Can the scroll view take focus itself?
`id`	Optional identifier.
`classes`	Optional classes.
`style`	A `style()`; by default the view fills its parent.

Details

Keys (when the scroll view or an unfocusable descendant has focus, or a focused child does not use the key itself): arrows scroll by one cell, Page Up / Page Down by a page, Home / End to the start / end. When focus moves to a widget inside a scroll view, the view scrolls to reveal it. Offset changes send a `"scroll.changed"` message.

Value

A `ScrollView` widget with methods `scroll_to(x, y)`, `scroll_by(dx, dy)`, `scroll_home()`, `scroll_end()` and `scroll_into_view(widget)`, and fields `offset_x`, `offset_y`, `viewport`, `virtual_size` and `max_scroll`.

Examples

```
items <- lapply(1:100, function(i) label(paste("Item", i)))
view <- scroll_view(items, id = "list")
view$scroll_to(y = 50)
```

`Select` *Select widget*

Description

A drop-down choice. See `dropdown()`.

Super class

`Widget` -> `Select`

Public fields

`focusable` Selects can be focused.

`prompt` Text shown when nothing is selected.

Active bindings

`value` The selected value or `NULL` (reactive; sends `"dropdown.changed"`).

`choices` The choice values.

Methods

Public methods:

- `Select$new()`
- `Select$default_style()`
- `Select$default_bindings()`
- `Select$open()`
- `Select$action_open()`
- `Select$on_click()`
- `Select$render()`
- `Select$clone()`

`Select$new()`: Create a select. See `dropdown()`.

Usage:

```
Select$new(
  choices,
  value = NULL,
  prompt = "Select",
  id = NULL,
  classes = NULL,
  style = NULL,
  disabled = FALSE
)
```

Arguments:

`choices` Choices.

`value` Initial value.

`prompt` Prompt.

`id`, `classes`, `style`, `disabled` See Widget.

`Select$default_style()`: The built-in style.

Usage:

```
Select$default_style()
```

`Select$default_bindings()`: Enter, Space and Down open the list.

Usage:

```
Select$default_bindings()
```

`Select$open()`: Open the drop-down list.

Usage:

```
Select$open()
```

`Select$action_open()`: Action used by the bindings.

Usage:

```
Select$action_open()
```

`Select$on_click()`: Clicking opens the list.

Usage:

```
Select$on_click(event)
```

Arguments:

`event` A `MouseEvent`.

`Select$render()`: The selected label and an arrow.

Usage:

```
Select$render()
```

`Select$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

set_timeout *Timers*

Description

Run a function later (`set_timeout()`) or repeatedly (`set_interval()`). Timers are part of the event loop: callbacks run between events, never concurrently, and never block the interface. Widgets have their own `widget$set_timeout()` / `widget$set_interval()`, whose timers are cancelled when the widget is removed.

Usage

```
set_timeout(delay, callback, app = current_app())

set_interval(interval, callback, app = current_app())
```

Arguments

`delay`, `interval`	Seconds.
`callback`	`function(app)`.
`app`	The app; defaults to the running app.

Value

A Timer; call `$cancel()` to stop it.

Examples

```
clock <- app(label("", id = "clock"), bind("q", "quit"))
clock$set_interval(1, function(app) {
  app$query_one("#clock")$update(format(Sys.time(), "%H:%M:%S"))
})
if (interactive()) run(clock)
```

signals *Reactive values, computed values and watchers*

Description

A small reactive graph for application state. `signal()` holds a value, `computed()` derives a value from signals (lazily, cached), and `watch()` runs code whenever the signals it reads change. Widgets can be created from functions that read signals - `label(function() ...)` - and are re-rendered automatically.

Usage

```
signal(value = NULL, name = NULL)

computed(fn, name = NULL)

watch(fn, name = NULL)

dispose(x)

batch(expr)

peek(x)

untracked(expr)

update_signal(x, fn)
```

Arguments

value	Initial value of a signal.
name	Optional graph label used in diagnostics and cycle paths.
fn	Function applied to the current value.
x	A signal or computed.
expr	An expression.

Details

- Read a signal with `x()`, write it with `x(value)`. Writing the same value (`identical()`) does nothing. Collections are replaced as a whole: `items(c(items(), "new"))` - changes made *inside* an object are not noticed.
- `computed(fn)` re-evaluates only when read after a dependency changed. A computed that depends on itself raises an error.
- `watch(fn)` runs `fn` now and again after its dependencies changed, in creation order. If it writes signals, the watchers depending on them run afterwards; a loop is stopped with an error. Use `dispose()` to stop it.
- `batch(expr)` defers watchers until `expr` is done, so several writes cause one run. Event handlers of an app run in a batch automatically.
- `peek(x)` reads a signal or computed without creating a dependency; `untracked(expr)` does the same for an expression.
- Watchers and bound widgets keep their signals' subscriber lists alive; call `dispose()` (or remove the widget) when they are no longer needed.

This graph is independent of the widget-level `reactive()` fields. It is **experimental**.

Value

`signal()` and `computed()` return functions (classes `termr_signal` / `termr_computed`); `watch()` returns a handle with a `$dispose()` method.

Examples

```
count <- signal(1)
double <- computed(function() count() * 2)
seen <- c()
w <- watch(function() seen <<- c(seen, double()))
count(2)
batch({ count(3); count(4) })
seen
dispose(w)
```

---

span *Styled text*

Description

`span()` creates a piece of text with its own style. Spans can be combined with `c()` and returned from a widget's `render` function or passed to `label()`. Newlines start new lines.

Usage

```
span(x = "", style = NULL, link = NULL)

## S3 method for class 'termr_text'
c(...)
```

Arguments

`x`	Text. A character vector is joined with newlines. May also be spans, to nest styles.
`style`	A `style()`; only colours and text attributes are used.
`link`	Optional link target (a URL or any string). Stored as metadata on the text for widgets that support it; not rendered yet.
`...`	Spans or character strings to combine.

Details

Spans nest: `span(c(span("a"), span("b", s1)), s2)` applies `s2` to both pieces, with the inner style taking precedence.

Value

An object of class `termr_text`.

Examples

```
c(span("Status: "), span("OK", style(foreground = "green", bold = TRUE)))
span(c(span("bold "), span("and red", style(foreground = "red"))), style(bold = TRUE))
```

`Sparkline` *Sparkline widget*

Description

See `sparkline()`.

Super class

`Widget` -> `Sparkline`

Public fields

`min, max` Fixed scale limits, or `NULL` to use the data range.

`max_length` Values kept by `push()`.

`summary` How to fit more values than columns: `"tail"` (latest values) or `"mean"` (average buckets).

Active bindings

`data` The values (reactive).

Methods

Public methods:

- `Sparkline$new()`
- `Sparkline$default_style()`
- `Sparkline$push()`
- `Sparkline$render_lines()`
- `Sparkline$content_width()`
- `Sparkline$clone()`

`Sparkline$new()`: Create a sparkline. See `sparkline()`.

Usage:

```
Sparkline$new(
  data = numeric(),
  min = NULL,
  max = NULL,
  summary = "tail",
  max_length = 1000L,
  id = NULL,
  classes = NULL,
  style = NULL
)
```

Arguments:

`data` Numbers.

`min`, `max` Scale limits.

`summary` Fitting strategy.

`max_length` Values kept by `push()`.

`id`, `classes`, `style` See Widget.

`Sparkline$default_style()`: The built-in style.

Usage:

```
Sparkline$default_style()
```

`Sparkline$push()`: Append values (keeping at most `max_length`).

Usage:

```
Sparkline$push(x)
```

Arguments:

`x` Numbers.

`Sparkline$render_lines()`: The bars for a given width.

Usage:

```
Sparkline$render_lines(width = NA_integer_)
```

Arguments:

`width` Content width.

`Sparkline$content_width()`: Natural width: one column per value.

Usage:

```
Sparkline$content_width()
```

`Sparkline$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`sparkline` *Sparkline*

Description

A tiny line chart made of block characters (`\u2581` to `\u2588`), one column per value. With more values than columns, the latest values are shown (`summary = "tail"`) or buckets are averaged (`"mean"`). Missing values are blank. Use `push()` to stream values.

Usage

```
sparkline(
  data = numeric(),
  min = NULL,
  max = NULL,
  summary = "tail",
  max_length = 1000L,
  id = NULL,
  classes = NULL,
  style = NULL
)
```

Arguments

`data`	Numeric vector.
`min`, `max`	Fixed scale; `NULL` uses the range of the data.
`summary`	`"tail"` or `"mean"`.
`max_length`	Number of values kept by `push()`.
`id`	Optional identifier.
`classes`	Optional classes.
`style`	A `style()`; the foreground colour colours the bars.

Value

A `Sparkline` with reactive `data` and method `push()`.

Examples

```
render_widget(sparkline(c(1, 4, 2, 8, 5, NA, 7)), 7, 1)
```

`Spinner` *Spinner widget*

Description

See `spinner()`.

Super class

`Widget` -> `Spinner`

Public fields

`paint_states` Animation frames only repaint.

`frames` Animation frames.

`interval` Seconds per frame.

Active bindings

`label` Text after the spinner (reactive).

`spinning` Is it animating? (reactive)

Methods

Public methods:

- `Spinner$new()`
- `Spinner$default_style()`
- `Spinner$on_mount()`
- `Spinner$render()`
- `Spinner$clone()`

`Spinner$new()`: Create a spinner. See `spinner()`.

Usage:

```
Spinner$new(
  label = "",
  type = "dots",
  interval = 0.1,
  id = NULL,
  classes = NULL,
  style = NULL
)
```

Arguments:

`label` Text after the spinner.

`type` Animation name.

`interval` Seconds per frame.

`id`, `classes`, `style` See Widget.

`Spinner$default_style()`: The built-in style.

Usage:

```
Spinner$default_style()
```

`Spinner$on_mount()`: Start the animation timer.

Usage:

```
Spinner$on_mount(event)
```

Arguments:

`event` A `MountEvent`.

`Spinner$render()`: The current frame and the label.

Usage:

```
Spinner$render()
```

`Spinner$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`spinner` *Spinner*

Description

A small animation showing that work is in progress. It runs on a timer of the widget, so it stops automatically when the widget is removed.

Usage

```
spinner(
  label = "",
  type = "dots",
  interval = 0.1,
  id = NULL,
  classes = NULL,
  style = NULL
)
```

Arguments

`label`	Text shown after the spinner.
`type`	`"dots"`, `"line"` or `"arc"` (ASCII `"line"` is used when the terminal cannot show Unicode).
`interval`	Seconds per frame.
`id`	Optional identifier.
`classes`	Optional classes.
`style`	A `style()`.

Value

A `Spinner` with reactive `label` and `spinning`.

Examples

```
spinner("Loading data...")
```

SplitPane *SplitPane widget*

Description

Two panes with a draggable divider. See `split_pane()`.

Super class

`Widget` -> `SplitPane`

Public fields

`direction` `"horizontal"` (panes side by side) or `"vertical"` (stacked).

`min_size` Smallest size of a pane in cells (when there is room).

Active bindings

`ratio` Share of the first pane (0 to 1). Assigning moves the divider (limited by `min_size`).

`sizes` The current sizes (cells) of the two panes.

Methods

Public methods:

- `SplitPane$new()`
- `SplitPane$default_style()`
- `SplitPane$arrange_children()`
- `SplitPane$move_divider()`
- `SplitPane$set_ratio()`
- `SplitPane$drag_to()`
- `SplitPane$announce()`
- `SplitPane$clone()`

`SplitPane$new()`: Create a split pane. See `split_pane()`.

Usage:

```
SplitPane$new(
  first,
  second,
  direction = "horizontal",
  ratio = 0.5,
  min_size = 3L,
  id = NULL,
  classes = NULL,
  style = NULL
)
```

Arguments:

`first, second` The two panes (widgets).

`direction, ratio, min_size` See `split_pane()`.

`id, classes, style` See Widget.

`SplitPane$default_style()`: The built-in style.

Usage:

```
SplitPane$default_style()
```

`SplitPane$arrange_children()`: Split the content area between the panes and the divider.

Usage:

```
SplitPane$arrange_children(children, inner, st)
```

Arguments:

`children` Visible children.

`inner` Content rectangle.

`st` Computed style.

`SplitPane$move_divider()`: Move the divider by some cells.

Usage:

```
SplitPane$move_divider(cells)
```

Arguments:

`cells` Positive: towards the second pane.

`SplitPane$set_ratio()`: Set the share of the first pane.

Usage:

```
SplitPane$set_ratio(ratio)
```

Arguments:

`ratio` A number between 0 and 1.

`SplitPane$drag_to()`: Place the divider under a screen position (drags).

Usage:

```
SplitPane$drag_to(x, y)
```

Arguments:

`x, y` Screen position.

`SplitPane$announce()`: Send `"splitpane.resized"` with the current ratio.

Usage:

```
SplitPane$announce()
```

`SplitPane$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

split_pane *Split pane*

Description

Two panes separated by a divider the user can drag with the mouse (press on the divider, move, release) or move with the keyboard (focus the divider with Tab; arrows move it, Shift+arrows ten cells, Home / End to the ends). Typical uses: a tree next to a preview, a table above a detail view, an editor above its output.

Usage

```
split_pane(
  first,
  second,
  direction = "horizontal",
  ratio = 0.5,
  min_size = 3L,
  id = NULL,
  classes = NULL,
  style = NULL
)
```

Arguments

first, second	The two panes.
direction	"horizontal": panes side by side, divider vertical; "vertical": stacked panes.
ratio	Share of the first pane, between 0 and 1.
min_size	Smallest pane size in cells.
id	Optional identifier.
classes	Optional classes.
style	A style().

Details

The panes fill their share of the space regardless of their own size styles. ratio is the share of the first pane; min_size keeps both panes from collapsing. The pane sends "splitpane.resized" (ratio) when the user finishes moving the divider. **Experimental.**

Value

A SplitPane widget with $ratio and $sizes.

Examples

```
ui <- split_pane(label("left"), label("right"), ratio = 0.4)
render_widget(ui, 20, 2)$to_text()
```

sql_editor *SQL editor widget*

Description

A thin text_area() subclass with SQL highlighting. When connection is supplied, execute_key runs the selected SQL, or the full buffer if nothing is selected. Queries run synchronously in the current R process.

Usage

```
sql_editor(
  value = "",
  id = NULL,
  connection = NULL,
  execute_key = "ctrl+enter",
  execution = c("query", "execute"),
  line_numbers = TRUE,
  tab_size = 2L,
  ...,
  result_mode = c("data", "lazy"),
  query_adapter = NULL
)
```

Arguments

value	Initial SQL text.
id	Optional widget id.
connection	A db_connection() wrapper or raw DBI connection.
execute_key	Key binding used to execute SQL.
execution	Either "query" (DBI::dbGetQuery) or "execute" (DBI::dbExecute).
line_numbers	Show line numbers? Defaults to TRUE.
tab_size	Spaces inserted by Tab.
...	Additional arguments passed to TextArea.
result_mode	Query result mode: "data" materializes the result (the default); "lazy" returns a db_query_source() for a paged table. Lazy mode is available only with execution = "query".
query_adapter	Optional backend pager callbacks passed to db_query_source().

Value

A TextArea subclass with an execute() method.

sql_highlighter *SQL syntax highlighter*

Description

A tolerant, line-oriented tokenizer for SQL text areas. `dialect` is reserved for small dialect-specific extensions; token classes are shared.

Usage

```
sql_highlighter(
  dialect = c("generic", "postgres", "sqlite", "mysql", "sqlserver")
)
```

Arguments

`dialect`	SQL dialect: `generic`, `postgres`, `sqlite`, `mysql`, or `sqlserver`.

Value

A highlighter function for `text_area()`.

StatusBar *StatusBar widget*

Description

See `status_bar()`.

Super class

`Widget` -> `StatusBar`

Public fields

`paint_states` Names of state fields whose changes only need a repaint (see `invalidate_paint()`); used by `set_state()`.

Active bindings

`left` Text in the left region.

`center` Text in the center region.

`right` Text in the right region.

Methods

Public methods:

- StatusBar$new()
- StatusBar$default_style()
- StatusBar$paint()
- StatusBar$clone()

`StatusBar$new()`: State fields are reactive and can be updated directly.
Create a status bar. See `status_bar()`.

Usage:

```
StatusBar$new(
  left = "",
  center = "",
  right = "",
  id = NULL,
  classes = NULL,
  style = NULL,
  disabled = FALSE
)
```

Arguments:

`left`, `center`, `right` Region text.

`id`, `classes`, `style` See Widget.

`disabled` Is the widget disabled?

`StatusBar$default_style()`: The built-in one-row style.

Usage:

```
StatusBar$default_style()
```

`StatusBar$paint()`: Paint the three text regions.

Usage:

```
StatusBar$paint(buffer, area, st)
```

Arguments:

`buffer` A ScreenBuffer.

`area` Visible rectangle.

`st` Computed style.

`StatusBar$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`status_bar` *One-line status bar*

Description

Displays left, centered and right status text in one terminal row. On a narrow screen it preserves the left region first, then the right region, and truncates the center region first. Truncation respects terminal cell widths and grapheme boundaries. Each value may be a no-argument reactive function that reads `signal()`s.

Usage

```
status_bar(
  left = "",
  center = "",
  right = "",
  id = NULL,
  classes = NULL,
  style = NULL,
  disabled = FALSE
)
```

Arguments

`left`, `center`, `right`	Text for the three regions.
`id`	Optional identifier.
`classes`	Optional classes.
`style`	A `style()`.
`disabled`	Disable the widget?

Value

A `StatusBar` widget with reactive `left`, `center`, and `right` fields.

Examples

```
status_bar("SQLite :memory:", "127 rows", "8 ms")
```

strip_ansi *Remove ANSI escape sequences from a string*

Description

Remove ANSI escape sequences from a string

Usage

```
strip_ansi(x)
```

Arguments

x	A character vector.

Value

x without CSI/OSC escape sequences.

Examples

```
strip_ansi("\033[1mbold\033[0m")
```

style *Widget styles*

Description

`style()` describes how a widget looks and how much space it takes. Only the properties you set are stored, so styles can be layered: a widget's built-in default style is overridden by the style passed to its constructor, and state styles (`focus`, `disabled`, ...) are applied on top while the widget is in that state.

Usage

```
style(
  width = NULL,
  height = NULL,
  min_width = NULL,
  max_width = NULL,
  min_height = NULL,
  max_height = NULL,
  margin = NULL,
  padding = NULL,
  border = NULL,
  border_color = NULL,
```

```
  align = NULL,
  valign = NULL,
  wrap = NULL,
  layout = NULL,
  foreground = NULL,
  background = NULL,
  bold = NULL,
  italic = NULL,
  underline = NULL,
  reverse = NULL,
  dim = NULL,
  strike = NULL,
  grid_columns = NULL,
  grid_rows = NULL,
  grid_gap = NULL,
  column_span = NULL,
  row_span = NULL,
  focus = NULL,
  hover = NULL,
  disabled = NULL,
  states = NULL
)
```

Arguments

`width`, `height`	Size, see above.
`min_width`, `max_width`, `min_height`, `max_height`	Size limits in cells.
`margin`, `padding`	Space outside / inside the border. One value (all sides), two values (vertical, horizontal) or four values (top, right, bottom, left).
`border`	Border type: `"none"`, `"ascii"`, `"single"`, `"round"`, `"double"`, `"heavy"` or `"blank"`.
`border_color`	Border colour (defaults to the foreground colour).
`align`	Horizontal alignment of text, or of children inside a container: `"left"`, `"center"` or `"right"`.
`valign`	Vertical alignment: `"top"`, `"middle"` or `"bottom"`.
`wrap`	How text that is wider than the widget is wrapped: `"none"` (clip, the default), `"word"` or `"char"`. Wrapped text makes `"auto"` heights grow.
`layout`	How a container arranges its children: `"vertical"` or `"horizontal"`.
`foreground`, `background`	Text and background colours.
`bold`, `italic`, `underline`, `reverse`, `dim`, `strike`	Text attributes.
`grid_columns`, `grid_rows`	Grid tracks for `layout = "grid"`: a number of equal columns, or a vector of sizes such as `c("auto", "1fr", 20)`. Rows default to `"auto"` and are added as needed.

`grid_gap` Space between grid cells: one value, or `c(rows, columns)`.

`column_span`, `row_span`
How many grid cells a child occupies.

`focus`, `hover`, `disabled`
Styles applied while the widget has focus, is under the mouse pointer, or is disabled.

`states` A named list of styles for other widget states, e.g. `list(pressed = style(reverse = TRUE))` for buttons.

Details

Sizes:

`width` and `height` accept:

- a number: a fixed number of cells;
- `"auto"`: the natural size of the content;
- `"<n>fr"`: a share of the remaining space (e.g. `"1fr"`, `"2fr"`);
- `"<n>%"`: a percentage of the parent's content area.

Colours:

`foreground`, `background` and `border_color` accept ANSI names (`"red"`, `"bright_blue"`, `"grey"`), integers 0-255 (xterm palette), hex strings (`"#ff8700"`) and R colour names (`"steelblue"`).

Value

An object of class `termr_style`.

Examples

```
style(width = 20, padding = c(0, 1), border = "round",
      foreground = "white", background = "blue", bold = TRUE,
      focus = style(background = "bright_blue"))
```

---

stylesheet *Stylesheets*

Description

Parse termr stylesheets (RTCSS, a small CSS-like language) to style many widgets at once by type, id, class and state. Pass them to `app(stylesheet = )` or `app$add_stylesheet()`.

Usage

```
stylesheet(text, source = "<stylesheet>")

stylesheet_file(path)
```

Arguments

text	Stylesheet text.
source	Name used in error messages.
path	Path to a stylesheet file.

Details

```
Button { border: round; padding: 0 2; }
Button.danger { background: $error; }
#save { width: 20; }
Input:focus { border: heavy $accent; }
Vertical > Label { bold: true; }
```

Properties are the arguments of style() (border-color and border_color both work). border also accepts a colour: border: round cyan. Colours can refer to the theme: $primary. Errors report the line, column and property.

Rules apply above a widget type's built-in style and below a widget's own style. More specific selectors win (ids over classes and pseudo classes over types); among equally specific rules, later ones win.

Value

A termr_stylesheet.

Examples

```
sheet <- stylesheet("Button { border: round; } #save { background: $primary; }")
sheet
```

---

table_filter *Create a DataTable column filter*

Description

Filters combine with AND in filter_columns(). Predicates can also be supplied directly as function(values) logical.

Usage

```
table_filter(
  type = c("equals", "contains", "regex", "range", "missing"),
  value = NULL,
  min = NULL,
  max = NULL,
  case_sensitive = FALSE,
  inclusive = TRUE
)
```

Arguments

`type`	One of `"equals"`, `"contains"`, `"regex"`, `"range"`, or `"missing"`.
`value`	Comparison value, text pattern, or regular expression.
`min`, `max`	Inclusive or exclusive range bounds.
`case_sensitive`	Case-sensitive text matching?
`inclusive`	Include range endpoints?

Value

An object suitable for a named `filter_columns()` list.

`table_source` *Create an experimental lazy table source*

Description

A table source lets `data_table()` request only the rows it needs. All callbacks are synchronous. `get_rows()` must return a data frame with the requested columns in the requested order. Row counts are known.

Usage

```
table_source(
  row_count,
  column_names,
  get_rows,
  sort = NULL,
  filter = NULL,
  search = NULL,
  row_key = NULL,
  set_value = NULL,
  refresh = NULL,
  close = NULL,
  on_error = NULL
)
```

Arguments

`row_count`	Function returning the current number of rows, or a non-negative scalar count.
`column_names`	Function returning unique, non-empty names, or a character vector.
`get_rows`	Function `function(start, count, columns = NULL)`.
`sort`, `filter`, `search`	Optional delegated operations.
`row_key`	Optional function mapping view positions to row keys.

`set_value`	Optional function `(row, column, value)` for editing.
`refresh, close`	Optional functions called by `$refresh()` and `$close()`.
`on_error`	Optional function returning a page-shaped placeholder when `get_rows()` fails. It receives the original condition and requested page.

Details

Optional `sort` and `filter` callbacks change the source's current view. They receive the sort keys or the named filter list used by `DataTable$sort()` and `DataTable$filter_columns()`. Returning `FALSE` rejects the operation; returning `NULL` or `TRUE` accepts it. A source should expose only operations it can implement without materializing its entire data set.

`search` receives a parsed query, column names, `start`, `direction`, and `include_current`; it returns `NULL` or `list(position, column)`. `row_key` maps view positions to stable row identities. `set_value(row, column, value)` receives a current view position. `refresh` and `close` are optional lifecycle callbacks. `on_error(error, start, count, columns)` may return a correctly shaped placeholder data frame when a page read fails; the table then displays the error instead of aborting a paint. Closing a source is explicit and never closes resources the source does not own.

Value

An experimental `termr_table_source` object.

`TabPane` *TabPane widget*

Description

The content of one tab. See `tab()`.

Super classes

`Widget` -> `Vertical` -> `TabPane`

Active bindings

`label` The tab label (reactive).

Methods

Public methods:

- `TabPane$new()`
- `TabPane$clone()`

`TabPane$new()`**:** Create a pane.

Usage:

```
TabPane$new(
  label,
  ...,
  id = NULL,
  classes = NULL,
  style = NULL,
  disabled = FALSE
)
```

Arguments:

`label` Tab label.

`...` Child widgets.

`id`, `classes`, `style`, `disabled` See Widget.

`TabPane$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`Tabs` *Tabs widget*

Description

Tabbed content. See `tabs()`.

Super class

`Widget` -> `Tabs`

Public fields

`focusable` The tab bar can take focus (Left/Right switch tabs).

Active bindings

`active` Index of the active tab (assign to switch).

`active_tab` The active TabPane, or `NULL`.

`tab_count` Number of tabs.

Methods

Public methods:

- Tabs$new()
- Tabs$default_style()
- Tabs$default_bindings()
- Tabs$add_tab()
- Tabs$remove_tab()
- Tabs$activate()
- Tabs$action_next_tab()
- Tabs$action_previous_tab()
- Tabs$on_mouse_down()
- Tabs$arrange_children()
- Tabs$child_clip()
- Tabs$content_width()
- Tabs$content_height()
- Tabs$paint()
- Tabs$clone()

`Tabs$new()`: Create tabs. See `tabs()`.

Usage:

```
Tabs$new(
  ...,
  active = 1L,
  id = NULL,
  classes = NULL,
  style = NULL,
  disabled = FALSE
)
```

Arguments:

`...` `tab()` panes.

`active` Index or id of the initially active tab.

`id`, `classes`, `style`, `disabled` See Widget.

`Tabs$default_style()`: The built-in style.

Usage:

```
Tabs$default_style()
```

`Tabs$default_bindings()`: Left/Right switch tabs when the bar is focused; Ctrl+PageUp / Ctrl+PageDown switch from anywhere inside.

Usage:

```
Tabs$default_bindings()
```

`Tabs$add_tab()`: Add a tab at the end.

Usage:

```
Tabs$add_tab(pane, activate = FALSE)
```

Arguments:

`pane` A `tab()` (or any widget, wrapped in a tab named after its id).
`activate` Make it the active tab?

`Tabs$remove_tab()`: Remove a tab.

Usage:

```
Tabs$remove_tab(which)
```

Arguments:

`which` Index or id of the tab.

`Tabs$activate()`: Show a tab.

Usage:

```
Tabs$activate(which)
```

Arguments:

`which` Index or id of the tab.

`Tabs$action_next_tab()`: Show the next tab.

Usage:

```
Tabs$action_next_tab()
```

`Tabs$action_previous_tab()`: Show the previous tab.

Usage:

```
Tabs$action_previous_tab()
```

`Tabs$on_mouse_down()`: Clicking a label shows its tab.

Usage:

```
Tabs$on_mouse_down(event)
```

Arguments:

`event` A `MouseEvent`.

`Tabs$arrange_children()`: Panes go below the tab bar.

Usage:

```
Tabs$arrange_children(children, inner, st)
```

Arguments:

`children` Visible children.
`inner` Content rectangle.
`st` Computed style.

`Tabs$child_clip()`: Panes are clipped below the tab bar.

Usage:

```
Tabs$child_clip(st)
```

Arguments:

`st` Computed style.

`Tabs$content_width()`: Natural width: the widest of the bar and the panes.

Usage:

```
Tabs$content_width()
```

`Tabs$content_height()`: Natural height: bar, rule and the active pane.

Usage:

```
Tabs$content_height(width)
```

Arguments:

`width` Content width.

`Tabs$paint()`: Paint the tab bar.

Usage:

```
Tabs$paint(buffer, area, st)
```

Arguments:

`buffer` A ScreenBuffer.

`area` Visible part of the region.

`st` Computed style.

`Tabs$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`tabs` *Tabs*

Description

`tabs()` shows one of several panes, with a tab bar on top. Only the active pane is laid out, painted and part of the focus chain. Switch tabs by clicking a label, with Left/Right while the tab bar has focus, or with Ctrl+PageUp / Ctrl+PageDown from anywhere inside. Switching sends a `"tabs.changed"` message (`event$data`: `index`, `previous`, `id`, `label`).

Usage

```
tabs(..., active = 1L, id = NULL, classes = NULL, style = NULL)

tab(label, ..., id = NULL, classes = NULL, style = NULL)
```

Arguments

`...`	`tab()` panes.
`active`	Index or id of the initially active tab.
`id`	Optional identifier.
`classes`	Optional classes.
`style`	A `style()`.
`label`	Tab label.

Value

A `Tabs` widget with methods `activate()`, `add_tab()`, `remove_tab()` and fields `active`, `active_tab`, `tab_count`.

Examples

```
ui <- tabs(
  tab("Overview", label("Summary")),
  tab("Logs", label("No logs yet"), id = "logs")
)
render_widget(ui, 30, 4)
```

`terminal_capabilities` *Terminal capabilities*

Description

What the terminal can do, detected once by the driver and consulted by everything above it. Detection is conservative: a feature that cannot be recognised from `TERM`, `COLORTERM`, `NO_COLOR`, `CI` or a few terminal-specific variables is reported as unavailable and termr falls back (e.g. to its internal clipboard instead of OSC 52).

Usage

```
terminal_capabilities(
  env = Sys.getenv(),
  windows = .Platform$OS.type == "windows",
  overrides = list()
)
```

Arguments

`env`	Environment variables (named character vector or list); defaults to the current ones.
`windows`	Is this a Windows console?
`overrides`	Named list of capabilities that replace detected ones.

Details

Fields of the returned list:

- `colors`: `"truecolor"`, `"256"`, `"16"` or `"none"`;
- `truecolor`, `unicode`, `mouse`, `sgr_mouse`, `bracketed_paste`, `alternate_screen`, `synchronized_output`, `cursor_shape`, `hyperlinks` (OSC 8), `osc52` (clipboard write): logical.

Overrides (all optional): `options(termr.color_mode = )`, `options(termr.ascii = TRUE)`, `options(termr.osc52 = TRUE/FALSE)`, `options(termr.hyperlinks = TRUE/FALSE)`, or the environment variables `TERMR_OSC52` and `TERMR_HYPERLINKS` set to `1` / `0`.

Value

An object of class `termr_capabilities`.

Examples

```
terminal_capabilities(env = c(TERM = "xterm-256color"), windows = FALSE)
terminal_capabilities(env = c(TERM = "dumb"), windows = FALSE)$mouse
```

---

termr_theme *Themes*

Description

A theme gives names to colours. Use them in styles and stylesheets as `"$name"`, e.g. `style(background = "$primary")`. Built-in widgets use theme colours, so a theme restyles a whole app: `app(ui, theme = "dark")` or `app$theme <- termr_theme("light", accent = "purple")`.

Usage

```
termr_theme(base = "default", ..., name = base)
```

Arguments

`base`	Name of a built-in theme to start from.
`...`	Colours to set, e.g. `primary = "#8a2be2"`.
`name`	Name of the theme.

Details

Named `termr_theme()` so that it does not mask `ggplot2::theme()`.

Colour names: `foreground`, `background` (screen background), `surface` (headers, panels), `muted` (secondary text, guides, disabled widgets), `primary` and `on_primary` (primary buttons, cursors), `accent` (focus), `success`, `warning`, `error`, `information`, `stripe` (zebra rows). Extra names can be added.

Built-in themes: "default" (the terminal's own 16-colour palette), "dark" and "light" (24-bit colours) and "high-contrast" (black and white with saturated yellow / cyan highlights and a heavier border around the focused widget).

On a terminal without colour (NO_COLOR, TERM=dumb) selections and cursors are drawn in reverse video, errors in bold underline and secondary text dimmed, so nothing depends on colour alone.

Value

A termr_theme.

Examples

```
termr_theme("dark", accent = "orange")
```

---

test_app *Test an app without a terminal*

Description

Starts the app with a headless driver and returns a Pilot that simulates the user. Every action runs one tick of the real event loop (input -> events -> timers -> repaint), and the output goes through the real renderer into a virtual terminal, so tests see exactly what a user would see.

Usage

```
test_app(app, width = 80L, height = 24L, color_mode = "truecolor")
```

Arguments

app	An App or a widget.
width, height	Virtual terminal size.
color_mode	Colour support of the virtual terminal: "truecolor" (default), "256", "16" or "none" (monochrome, as with NO_COLOR).

Details

Pilot methods:

- press(...): press keys, e.g. press("tab", "enter", "ctrl+c");
- type(text): type characters;
- resize(width, height): resize the terminal;
- advance(seconds): let simulated time pass (fires timers);
- click(target, y = NULL, button = "left"): click a widget (a selector or widget; the centre of its visible part is used) or a position click(x, y);
- hover(target), scroll(target, direction = "down", times = 1): move the pointer, turn the mouse wheel;

- `mouse(action, x, y, button, direction)`: send a raw mouse event;
- `drag(from, to, button = "left", steps = 3)`: press, move and release between two positions (selectors, widgets or `c(x, y)`);
- `paste(text)`: deliver a bracketed paste; `system_clipboard()`: text of the last OSC 52 write;
- `focus(target)`, `find(selector)`: focus a widget, find one;
- `wait_for(condition, timeout = 5)`: step the event loop (simulated time, real time while workers run) until `condition(app)` is `TRUE`;
- `run_worker(fn, ..., wait = TRUE)`: start a worker and wait for it;
- `wait_for_workers(timeout)`: wait for background workers to finish;
- `screen_text()`: the visible screen as a character vector;
- `snapshot()`: print the screen (for `testthat::expect_snapshot()`);
- `query_one(selector)`, `query(selector)`: find widgets;
- `stop()`: exit the app and restore the (virtual) terminal.

Fields: `app`, `driver`, `exited` and `value` (the value passed to `app$exit()`).

Value

A `Pilot` object.

Examples

```
pilot <- test_app(app(input(id = "name")), width = 30, height = 5)
pilot$type("Ada")
pilot$query_one("#name")$value
pilot$stop()
```

`TextArea` *TextArea widget*

Description

A multi-line text editor. See `text_area()`.

Super class

`Widget` -> `TextArea`

Public fields

`focusable` Text areas can be focused.

`placeholder` Text shown while the area is empty.

`line_numbers` Show a line number gutter?

`wrap` Wrap long lines (`TRUE`) or scroll horizontally?

`read_only` Block editing (cursor, selection, copy and search still work)?

`language` Optional language name passed to the configured highlighter.

`tab_size` Spaces inserted by Tab (and used for tabs in loaded text).

`tab_behavior` `"focus"`: Tab moves focus (default, safe in forms); `"indent"`: Tab inserts spaces and Shift+Tab removes them.

`auto_indent` Copy the indentation of the current line on Enter?

`highlighter` Optional function `(lines, state)` returning a list of per-line data frames with 1-based inclusive `start`, `end`, `token`.

`validate` `NULL` or `function(value)` returning `NULL` (valid) or an error message.

Active bindings

`value` The text. Assigning replaces it (see `set_text()`).

`text` The text (read-only alias of `value`).

`lines` The lines as a character vector.

`n_lines` Number of lines.

`cursor_row, cursor_column` Cursor position, 1-based (column 1 is before the first character).

`selection` The selected text (`""` if none).

`can_undo, can_redo` Is there something to undo / redo?

`history_size` Characters held by the undo history.

`match_count` Number of matches of the current search.

`find_open` Is the find bar open?

`valid` Does the value pass `validate`?

`error` The validation message, or `NULL`.

Methods

Public methods:

- `TextArea$new()`
- `TextArea$default_style()`
- `TextArea$pseudo_states()`
- `TextArea$default_bindings()`
- `TextArea$on_key()`
- `TextArea$on_paste()`
- `TextArea$on_mouse_down()`

- TextArea$on_mouse_move()
- TextArea$on_mouse_up()
- TextArea$on_mouse_scroll()
- TextArea$set_text()
- TextArea$insert()
- TextArea$select_range()
- TextArea$goto_line()
- TextArea$cursor_position()
- TextArea$copy()
- TextArea$cut()
- TextArea$undo()
- TextArea$redo()
- TextArea$find()
- TextArea$find_next()
- TextArea$find_previous()
- TextArea$replace()
- TextArea$replace_all()
- TextArea$clear_find()
- TextArea$action_cursor_left()
- TextArea$action_cursor_right()
- TextArea$action_cursor_up()
- TextArea$action_cursor_down()
- TextArea$action_line_start()
- TextArea$action_line_end()
- TextArea$action_document_start()
- TextArea$action_document_end()
- TextArea$action_word_left()
- TextArea$action_word_right()
- TextArea$action_page_up()
- TextArea$action_page_down()
- TextArea$action_select_left()
- TextArea$action_select_right()
- TextArea$action_select_up()
- TextArea$action_select_down()
- TextArea$action_select_line_start()
- TextArea$action_select_line_end()
- TextArea$action_select_document_start()
- TextArea$action_select_document_end()
- TextArea$action_select_word_left()
- TextArea$action_select_word_right()
- TextArea$action_select_page_up()
- TextArea$action_select_page_down()

- TextArea$action_select_all()
- TextArea$action_newline()
- TextArea$action_delete_left()
- TextArea$action_delete_right()
- TextArea$action_delete_word_left()
- TextArea$action_delete_word_right()
- TextArea$action_delete_to_line_start()
- TextArea$action_delete_to_line_end()
- TextArea$action_paste()
- TextArea$action_indent()
- TextArea$action_dedent()
- TextArea$action_undo()
- TextArea$action_redo()
- TextArea$action_open_find()
- TextArea$action_find_next()
- TextArea$action_find_previous()
- TextArea$action_open_replace()
- TextArea$action_open_goto()
- TextArea$content_width()
- TextArea$content_height()
- TextArea$paint()
- TextArea$clone()

`TextArea$new()`: Create a text area.

Usage:

```
TextArea$new(
  value = "",
  placeholder = "",
  id = NULL,
  classes = NULL,
  style = NULL,
  disabled = FALSE,
  line_numbers = FALSE,
  wrap = TRUE,
  read_only = FALSE,
  language = NULL,
  tab_size = 4L,
  tab_behavior = "focus",
  auto_indent = FALSE,
  validate = NULL,
  max_history = 200L,
  highlighter = NULL
)
```

Arguments:

`value` Initial text.

`placeholder` Placeholder text.

`id`, `classes`, `style`, `disabled` See Widget.

`line_numbers`, `wrap`, `read_only`, `language`, `tab_size`, `tab_behavior`, `auto_indent` See the fields.

`validate` Validation function.

`max_history` Maximum number of undo steps.

`highlighter` Optional function `(lines, state)` returning a list of per-line data frames with 1-based inclusive `start`, `end`, `token`.

`TextArea$default_style()`: The built-in style.

Usage:

```
TextArea$default_style()
```

`TextArea$pseudo_states()`: Active states: focus, hover, disabled and invalid.

Usage:

```
TextArea$pseudo_states()
```

`TextArea$default_bindings()`: Editing key bindings.

Usage:

```
TextArea$default_bindings()
```

`TextArea$on_key()`: Printable keys replace the selection or are inserted at the cursor; Ctrl+C / Ctrl+X copy / cut a selection. While the find bar is open, keys edit the search text.

Usage:

```
TextArea$on_key(event)
```

Arguments:

`event` A `KeyEvent`.

`TextArea$on_paste()`: Pasted text is inserted as one undo step; line breaks are kept.

Usage:

```
TextArea$on_paste(event)
```

Arguments:

`event` A `PasteEvent`.

`TextArea$on_mouse_down()`: Click to place the cursor, shift+click and drag to select.

Usage:

```
TextArea$on_mouse_down(event)
```

Arguments:

`event` A `MouseEvent`.

`TextArea$on_mouse_move()`: Dragging with the left button extends the selection.

Usage:

```
TextArea$on_mouse_move(event)
```

Arguments:

`event` A `MouseEvent`.

`TextArea$on_mouse_up()`: Ends a drag selection.

Usage:

```
TextArea$on_mouse_up(event)
```

Arguments:

`event` A `MouseEvent`.

`TextArea$on_mouse_scroll()`: The mouse wheel scrolls three rows.

Usage:

```
TextArea$on_mouse_scroll(event)
```

Arguments:

`event` A `MouseEvent`.

`TextArea$set_text()`: Replace the whole text. Clears the undo history and moves the cursor to the start.

Usage:

```
TextArea$set_text(text)
```

Arguments:

`text` A string (or character vector of lines).

`TextArea$insert()`: Insert text at the cursor, replacing the selection.

Usage:

```
TextArea$insert(text, kind = NULL)
```

Arguments:

`text` A string; line breaks are kept.

`kind` Internal: `"type"` merges into undo groups, `"paste"` does not.

`TextArea$select_range()`: Select a range (1-based rows and columns; column 1 is before the first character).

Usage:

```
TextArea$select_range(start_row, start_column, end_row, end_column)
```

Arguments:

`start_row`, `start_column`, `end_row`, `end_column` The range.

`TextArea$goto_line()`: Move the cursor to a line.

Usage:

```
TextArea$goto_line(row, column = 1L)
```

Arguments:

`row` Line number (1-based).

`column` Column (1-based).

`TextArea$cursor_position()`: Cursor position. Columns count grapheme clusters and terminal display cells, both 1-based.

Usage:

```
TextArea$cursor_position()
```

`TextArea$copy()`: Copy the selection to the app clipboard (and the system clipboard via OSC 52 when `system = TRUE` and supported).

Usage:

```
TextArea$copy(system = TRUE)
```

Arguments:

`system` Also write to the system clipboard?

`TextArea$cut()`: Copy the selection and delete it.

Usage:

```
TextArea$cut(system = TRUE)
```

Arguments:

`system` Also write to the system clipboard?

`TextArea$undo()`: Undo the last edit group.

Usage:

```
TextArea$undo()
```

`TextArea$redo()`: Redo the last undone edit.

Usage:

```
TextArea$redo()
```

`TextArea$find()`: Search for text and select the first match at or after the cursor.

Usage:

```
TextArea$find(query, case_sensitive = FALSE, regex = FALSE)
```

Arguments:

`query` Text or regular expression (PCRE).

`case_sensitive` Match case?

`regex` Is `query` a regular expression? Invalid expressions raise an error.

Returns: `TRUE` if there is a match (invisibly).

`TextArea$find_next()`: Select the next match (wrapping around).

Usage:

```
TextArea$find_next()
```

`TextArea$find_previous()`: Select the previous match (wrapping around).

Usage:

```
TextArea$find_previous()
```

`TextArea$replace()`: Replace the current search match, or find `query` and replace its next match.

Usage:

```
TextArea$replace(
  replacement,
  query = NULL,
  case_sensitive = FALSE,
  regex = FALSE
)
```

Arguments:

`replacement` Replacement text.

`query` Optional search text; defaults to the active search.

`case_sensitive, regex` Search options used when `query` is given.

`TextArea$replace_all()`: Replace all matches in one document edit. If the edit is larger than the bounded undo budget, it is applied without retaining an undo record.

Usage:

```
TextArea$replace_all(query, replacement, case_sensitive = FALSE, regex = FALSE)
```

Arguments:

`query` Search text or regular expression.

`replacement` Replacement text.

`case_sensitive` Match case?

`regex` Interpret `query` as a regular expression?

`TextArea$clear_find()`: Remove the search highlights and close the find bar.

Usage:

```
TextArea$clear_find()
```

`TextArea$action_cursor_left()`: Move one character left.

Usage:

```
TextArea$action_cursor_left()
```

`TextArea$action_cursor_right()`: Move one character right.

Usage:

```
TextArea$action_cursor_right()
```

`TextArea$action_cursor_up()`: Move one row up.

Usage:

```
TextArea$action_cursor_up()
```

`TextArea$action_cursor_down()`: Move one row down.

Usage:

```
TextArea$action_cursor_down()
```

`TextArea$action_line_start()`: Move to the start of the line.

Usage:

```
TextArea$action_line_start()
```

`TextArea$action_line_end()`: Move to the end of the line.

Usage:

```
TextArea$action_line_end()
```

`TextArea$action_document_start()`: Move to the start of the text.

Usage:

```
TextArea$action_document_start()
```

`TextArea$action_document_end()`: Move to the end of the text.

Usage:

```
TextArea$action_document_end()
```

`TextArea$action_word_left()`: Move to the previous word.

Usage:

```
TextArea$action_word_left()
```

`TextArea$action_word_right()`: Move to the next word.

Usage:

```
TextArea$action_word_right()
```

`TextArea$action_page_up()`: Move one page up.

Usage:

```
TextArea$action_page_up()
```

`TextArea$action_page_down()`: Move one page down.

Usage:

```
TextArea$action_page_down()
```

`TextArea$action_select_left()`: Extend the selection left.

Usage:

```
TextArea$action_select_left()
```

`TextArea$action_select_right()`: Extend the selection right.

Usage:

```
TextArea$action_select_right()
```

`TextArea$action_select_up()`: Extend the selection up.

Usage:

```
TextArea$action_select_up()
```

`TextArea$action_select_down()`: Extend the selection down.

Usage:

```
TextArea$action_select_down()
```

`TextArea$action_select_line_start()`: Select to the start of the line.

Usage:

```
TextArea$action_select_line_start()
```

`TextArea$action_select_line_end()`: Select to the end of the line.

Usage:

```
TextArea$action_select_line_end()
```

`TextArea$action_select_document_start()`: Select to the start of the text.

Usage:

```
TextArea$action_select_document_start()
```

`TextArea$action_select_document_end()`: Select to the end of the text.

Usage:

```
TextArea$action_select_document_end()
```

`TextArea$action_select_word_left()`: Select to the previous word.

Usage:

```
TextArea$action_select_word_left()
```

`TextArea$action_select_word_right()`: Select to the next word.

Usage:

```
TextArea$action_select_word_right()
```

`TextArea$action_select_page_up()`: Extend the selection one page up.

Usage:

```
TextArea$action_select_page_up()
```

`TextArea$action_select_page_down()`: Extend the selection one page down.

Usage:

```
TextArea$action_select_page_down()
```

`TextArea$action_select_all()`: Select everything.

Usage:

```
TextArea$action_select_all()
```

`TextArea$action_newline()`: Insert a line break (keeping the indentation when `auto_indent` is on).

Usage:

```
TextArea$action_newline()
```

`TextArea$action_delete_left()`: Delete the selection or the character before the cursor.

Usage:

```
TextArea$action_delete_left()
```

`TextArea$action_delete_right()`: Delete the selection or the character at the cursor.

Usage:

```
TextArea$action_delete_right()
```

`TextArea$action_delete_word_left()`: Delete the word before the cursor.

Usage:

```
TextArea$action_delete_word_left()
```

`TextArea$action_delete_word_right()`: Delete the word after the cursor.

Usage:

```
TextArea$action_delete_word_right()
```

`TextArea$action_delete_to_line_start()`: Delete from the start of the line to the cursor.

Usage:

```
TextArea$action_delete_to_line_start()
```

`TextArea$action_delete_to_line_end()`: Delete from the cursor to the end of the line (or the line break when the cursor is at the end).

Usage:

```
TextArea$action_delete_to_line_end()
```

`TextArea$action_paste()`: Insert the app clipboard.

Usage:

```
TextArea$action_paste()
```

`TextArea$action_indent()`: Insert spaces up to the next tab stop (or indent the selected lines).

Usage:

```
TextArea$action_indent()
```

`TextArea$action_dedent()`: Remove up to one tab stop of leading spaces.

Usage:

```
TextArea$action_dedent()
```

`TextArea$action_undo()`: Undo.

Usage:

```
TextArea$action_undo()
```

`TextArea$action_redo()`: Redo.

Usage:

```
TextArea$action_redo()
```

`TextArea$action_open_find()`: Open the find bar (Ctrl+F).

Usage:

```
TextArea$action_open_find()
```

`TextArea$action_find_next()`: Select the next match (F3).

Usage:

```
TextArea$action_find_next()
```

`TextArea$action_find_previous()`: Select the previous match (Shift+F3).

Usage:

```
TextArea$action_find_previous()
```

`TextArea$action_open_replace()`: Open the find and replace bar (Ctrl+H).

Usage:

```
TextArea$action_open_replace()
```

`TextArea$action_open_goto()`: Open the go-to-line bar (Ctrl+G).

Usage:

```
TextArea$action_open_goto()
```

`TextArea$content_width()`: Natural width: the longest line (at most 120 columns).

Usage:

```
TextArea$content_width()
```

`TextArea$content_height()`: Natural height: the number of lines (at most 1000).

Usage:

```
TextArea$content_height(width)
```

Arguments:

`width` Content width.

`TextArea$paint()`: Draw the visible part of the text.

Usage:

```
TextArea$paint(buffer, area, st)
```

Arguments:

`buffer` A ScreenBuffer.

`area` Visible part of the region.

`st` Computed style.

`TextArea$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`text_area` *Multi-line text editor*

Description

An editable text area with a cursor, selection, undo/redo, search and scrolling. Text is stored as a vector of lines and edited by logical (row, grapheme) positions, so emoji, accents and CJK text are never split, and only the visible lines are formatted when drawing.

Usage

```
text_area(
  value = "",
  placeholder = "",
  id = NULL,
  classes = NULL,
  style = NULL,
  disabled = FALSE,
  line_numbers = FALSE,
  wrap = TRUE,
  read_only = FALSE,
  language = NULL,
  tab_size = 4L,
  tab_behavior = "focus",
  auto_indent = FALSE,
  validate = NULL,
  max_history = 200L,
  highlighter = NULL
)
```

Arguments

`value`	Initial text.
`placeholder`	Text shown while the area is empty.
`id`	Optional identifier.
`classes`	Optional classes.
`style`	A `style()`.
`disabled`	Disabled areas cannot be focused or edited.
`line_numbers`	Show line numbers?
`wrap`	Wrap long lines? If `FALSE` the view scrolls horizontally.
`read_only`	Block editing?
`language`	Optional language label supplied to the highlighter.
`tab_size`	Spaces per tab stop.
`tab_behavior`	`"focus"` (Tab moves focus) or `"indent"`.

auto_indent	Keep the indentation on Enter?
validate	Optional function(value) returning NULL/TRUE when valid or a message.
max_history	Maximum number of undo steps.
highlighter	Optional viewport-first syntax highlighter; see r_highlighter().

Details

Keys: arrows, Home/End, Ctrl+Home/End, Ctrl+Left/Right (words), PageUp/PageDown, Shift+ to select, Ctrl+A select all, Enter, Backspace, Delete, Ctrl+Backspace / Ctrl+Delete (words), Ctrl+U / Ctrl+K (to line start / end), Ctrl+C / Ctrl+X / Ctrl+V (copy, cut, paste with the app clipboard; copy also reaches the system clipboard via OSC 52 when the terminal supports it), Ctrl+Z undo, Ctrl+Y redo, Ctrl+F find (Enter / Down next, Up previous, Ctrl+R regex, Alt+C case, Esc close), F3 / Shift+F3 next / previous match. Tab moves focus unless tab_behavior = "indent". The mouse places the cursor, drags select and the wheel scrolls. A bracketed paste is one undo step.

Messages (event$data): "textarea.changed" (version, lines, valid; read the text with $value), "textarea.selection_changed" (empty, start_row, start_column, end_row, end_column) and "textarea.valid" / "textarea.invalid" (error).

Editing groups: typing is merged into undo steps per word, paste, delete-selection and line operations are single steps. The history is bounded (max_history steps and about 2 million characters). Tabs in text are expanded to spaces. This widget is **experimental**.

Value

A TextArea widget. Useful members: $value, $set_text(), $insert(), $selection, $cursor_row, $cursor_column, $goto_line(), $cursor_position(), $undo(), $redo(), $find(), $replace(), $replace_all(), $find_next(), $find_previous(), $match_count.

Examples

```
ed <- text_area("Hello\nworld", id = "editor", line_numbers = TRUE)
ed$insert("!")
ed$value
ed$find("wor")
ed$selection
```

Timer *Timer handle*

Description

Returned by set_timeout() and set_interval().

Public fields

`delay` Delay (or interval) in seconds.

`repeating` Does the timer repeat?

`due` Clock time of the next run.

`callback` Function called when the timer fires.

`active` Is the timer still scheduled?

`runs` How many times the timer has fired.

Methods

Public methods:

- `Timer$new()`
- `Timer$cancel()`
- `Timer$print()`
- `Timer$clone()`

`Timer$new()`: Create a timer (use `set_timeout()` instead).

Usage:

```
Timer$new(delay, callback, repeating, now)
```

Arguments:

`delay` Seconds.

`callback` Function.

`repeating` Logical.

`now` Current clock time.

`Timer$cancel()`: Stop the timer.

Usage:

```
Timer$cancel()
```

`Timer$print()`: Print the timer.

Usage:

```
Timer$print(...)
```

Arguments:

`...` Ignored.

`Timer$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Toast *Toast widget*

Description

A notification shown by `app$notify()`.

Super classes

`Widget` -> `Label` -> `Toast`

Public fields

`severity` `"information"`, `"success"`, `"warning"` or `"error"`.

Methods

Public methods:

- `Toast$new()`
- `Toast$default_style()`
- `Toast$on_click()`
- `Toast$clone()`

`Toast$new()`: Create a toast (use `app$notify()`).

Usage:

```
Toast$new(message, title = NULL, severity = "information")
```

Arguments:

`message` Text.

`title` Optional title.

`severity` Severity.

`Toast$default_style()`: The built-in style.

Usage:

```
Toast$default_style()
```

`Toast$on_click()`: Clicking closes the toast.

Usage:

```
Toast$on_click(event)
```

Arguments:

`event` A `MouseEvent`.

`Toast$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`TreeNode` *Tree node*

Description

A node of a tree shown by `tree_view()`. See `tree_node()`.

Public fields

`label` Text shown for the node.

`data` Any R value attached to the node.

`id` Optional identifier.

`loader` `NULL` or `function(node)` returning children, called the first time the node is expanded.

`loaded` Has the loader run?

Active bindings

`parent` The parent node (read-only).

`children` Child nodes (read-only).

`expanded` Is the node expanded? (assign to expand / collapse)

Methods

Public methods:

- `TreeNode$new()`
- `TreeNode$add()`
- `TreeNode$remove()`
- `TreeNode$clear()`
- `TreeNode$expand()`
- `TreeNode$collapse()`
- `TreeNode$toggle()`
- `TreeNode$expandable()`
- `TreeNode$depth()`
- `TreeNode$ancestors()`
- `TreeNode$path()`
- `TreeNode$walk()`
- `TreeNode$find()`
- `TreeNode$changed()`
- `TreeNode$print()`
- `TreeNode$clone()`

`TreeNode$new()`**:** Create a node. See `tree_node()`.

Usage:

```
TreeNode$new(
  label,
  children = list(),
  data = NULL,
  expanded = FALSE,
  id = NULL,
  loader = NULL
)
```

Arguments:

`label` Label.

`children` Child nodes (or labels).

`data` Attached value.

`expanded` Start expanded?

`id` Identifier.

`loader` Lazy loader.

`TreeNode$add()`: Add a child.

Usage:

```
TreeNode$add(node, data = NULL, expanded = FALSE)
```

Arguments:

`node` A `TreeNode` or a label.

`data` Data for a node created from a label.

`expanded` Expanded state for a node created from a label.

Returns: The child node.

`TreeNode$remove()`: Detach the node from its parent.

Usage:

```
TreeNode$remove()
```

`TreeNode$clear()`: Remove all children.

Usage:

```
TreeNode$clear()
```

`TreeNode$expand()`: Expand the node (running the loader if needed).

Usage:

```
TreeNode$expand()
```

`TreeNode$collapse()`: Collapse the node.

Usage:

```
TreeNode$collapse()
```

`TreeNode$toggle()`: Toggle between expanded and collapsed.

Usage:

```
TreeNode$toggle()
```

`TreeNode$expandable()`: Can the node be expanded?

Usage:

```
TreeNode$expandable()
```

`TreeNode$depth()`: Depth below the root (the root is 0).

Usage:

```
TreeNode$depth()
```

`TreeNode$ancestors()`: Parent, grandparent, ... up to the root.

Usage:

```
TreeNode$ancestors()
```

`TreeNode$path()`: Labels from the root to this node.

Usage:

```
TreeNode$path()
```

`TreeNode$walk()`: This node and all descendants (depth first).

Usage:

```
TreeNode$walk()
```

`TreeNode$find()`: Find a descendant by id.

Usage:

```
TreeNode$find(id)
```

Arguments:

`id` Identifier.

`TreeNode$changed()`: Notify the tree view that the node changed.

Usage:

```
TreeNode$changed()
```

`TreeNode$print()`: Print the subtree.

Usage:

```
TreeNode$print(...)
```

Arguments:

`...` Ignored.

`TreeNode$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

TreeView *TreeView widget*

Description

Shows a tree of tree_node()s. See tree_view().

Super class

Widget -> TreeView

Public fields

paint_states Cursor and scroll changes only repaint.

focusable Trees can be focused.

show_root Show the root node?

guides Draw guide lines?

Active bindings

root The root node.

cursor_node The node under the cursor, or NULL.

visible_count Number of visible (expanded) lines.

offset First visible line minus one.

Methods

Public methods:

- TreeView$new()
- TreeView$default_style()
- TreeView$default_bindings()
- TreeView$tree_changed()
- TreeView$select()
- TreeView$action_cursor_up()
- TreeView$action_cursor_down()
- TreeView$action_page_up()
- TreeView$action_page_down()
- TreeView$action_first()
- TreeView$action_last()
- TreeView$action_expand_or_child()
- TreeView$action_collapse_or_parent()
- TreeView$action_toggle()
- TreeView$action_activate()

- TreeView$on_mouse_down()
- TreeView$on_mouse_scroll()
- TreeView$content_width()
- TreeView$content_height()
- TreeView$paint()
- TreeView$clone()

`TreeView$new()`: Create a tree view. See `tree_view()`.

Usage:

```
TreeView$new(
  root,
  show_root = TRUE,
  guides = TRUE,
  id = NULL,
  classes = NULL,
  style = NULL,
  disabled = FALSE
)
```

Arguments:

`root` The root TreeNode.

`show_root`, `guides` See fields.

`id`, `classes`, `style`, `disabled` See Widget.

`TreeView$default_style()`: The built-in style.

Usage:

```
TreeView$default_style()
```

`TreeView$default_bindings()`: Navigation keys.

Usage:

```
TreeView$default_bindings()
```

`TreeView$tree_changed()`: Called by nodes when the tree changes.

Usage:

```
TreeView$tree_changed()
```

`TreeView$select()`: Move the cursor to a node (expanding its ancestors).

Usage:

```
TreeView$select(node)
```

Arguments:

`node` A node of this tree.

`TreeView$action_cursor_up()`: Move up.

Usage:

```
TreeView$action_cursor_up()
```

`TreeView$action_cursor_down()`: Move down.

Usage:

```
TreeView$action_cursor_down()
```

`TreeView$action_page_up()`: Move a page up.

Usage:

```
TreeView$action_page_up()
```

`TreeView$action_page_down()`: Move a page down.

Usage:

```
TreeView$action_page_down()
```

`TreeView$action_first()`: Go to the first node.

Usage:

```
TreeView$action_first()
```

`TreeView$action_last()`: Go to the last visible node.

Usage:

```
TreeView$action_last()
```

`TreeView$action_expand_or_child()`: Expand, or go to the first child.

Usage:

```
TreeView$action_expand_or_child()
```

`TreeView$action_collapse_or_parent()`: Collapse, or go to the parent.

Usage:

```
TreeView$action_collapse_or_parent()
```

`TreeView$action_toggle()`: Toggle the cursor node.

Usage:

```
TreeView$action_toggle()
```

`TreeView$action_activate()`: Toggle and send `"tree.node_activated"`.

Usage:

```
TreeView$action_activate()
```

`TreeView$on_mouse_down()`: Clicking selects; clicking the arrow toggles.

Usage:

```
TreeView$on_mouse_down(event)
```

Arguments:

`event` A `MouseEvent`.

`TreeView$on_mouse_scroll()`: The wheel scrolls three lines.

Usage:

```
TreeView$on_mouse_scroll(event)
```

Arguments:

`event` A `MouseEvent`.

`TreeView$content_width()`: Natural width: the widest visible line.

Usage:

```
TreeView$content_width()
```

`TreeView$content_height()`: Natural height: visible lines (at most 1000).

Usage:

```
TreeView$content_height(width)
```

Arguments:

`width` Content width.

`TreeView$paint()`: Draw the visible lines.

Usage:

```
TreeView$paint(buffer, area, st)
```

Arguments:

`buffer` A ScreenBuffer.

`area` Visible part of the region.

`st` Computed style.

`TreeView$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`tree_node` *Tree nodes*

Description

Tree nodes

Usage

```
tree_node(label, ..., data = NULL, expanded = FALSE, id = NULL, loader = NULL)
```

Arguments

label	Text shown for the node.
...	Child nodes, or strings (leaf nodes).
data	Any R value attached to the node (e.g. a file path).
expanded	Start expanded?
id	Optional identifier (see node$find(id)).
loader	Optional function(node) returning child nodes (or labels), called the first time the node is expanded. Use it for large or expensive trees such as file systems.

Value

A TreeNode.

Examples

```
root <- tree_node("Project", tree_node("R", "app.R", "widget.R"), "README.md", expanded = TRUE)
root
```

---

tree_view *Tree view*

Description

Shows a tree_node() hierarchy with expand/collapse arrows and guide lines. Only visible lines are painted, and children can be loaded lazily.

Usage

```
tree_view(
  root,
  show_root = TRUE,
  guides = TRUE,
  id = NULL,
  classes = NULL,
  style = NULL
)
```

Arguments

root	The root tree_node().
show_root	Show the root itself (otherwise its children are the top level and it is always expanded).
guides	Draw guide lines?
id	Optional identifier.
classes	Optional classes.
style	A style().

Details

Keys: Up/Down/Page Up/Page Down/Home/End move; Right expands (or goes to the first child); Left collapses (or goes to the parent); Space toggles; Enter toggles and sends `"tree.node_activated"`. Clicking selects a node; clicking its arrow toggles it.

Messages (`event$data`: `node`, `label`, `data`, `id`): `"tree.node_selected"`, `"tree.node_expanded"`, `"tree.node_collapsed"`, `"tree.node_activated"`.

Value

A `TreeView` widget with method `select(node)` and fields `root`, `cursor_node`, `visible_count`.

Examples

```
root <- tree_node("Project", tree_node("R", "app.R", "widget.R", expanded = TRUE),
                  "README.md", expanded = TRUE)
render_widget(tree_view(root), 24, 5)
```

---

unregister_layout *Remove an experimental custom layout*

Description

Built-in layouts cannot be removed.

Usage

```
unregister_layout(name)
```

Arguments

`name`	A registered `custom_` layout name.

Value

`TRUE` when the layout was removed.

Vertical *Vertical container*

Description

Arranges children top to bottom. See `vertical()`.

Super class

`Widget` -> `Vertical`

Methods

Public methods:

- `Vertical$default_style()`
- `Vertical$clone()`

`Vertical$default_style()`: The built-in style.

Usage:

```
Vertical$default_style()
```

`Vertical$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

vertical *Layout containers*

Description

`vertical()` stacks its children top to bottom; `horizontal()` places them side by side. By default a vertical container fills the available space, and a horizontal container is as tall as its tallest child.

Usage

```
vertical(..., id = NULL, classes = NULL, style = NULL)

horizontal(..., id = NULL, classes = NULL, style = NULL)
```

Arguments

`...`	Child widgets.
`id`	Optional identifier (for `#id` selectors).
`classes`	Optional classes (for `.class` selectors).
`style`	A `style()`.

Details

Children sized `"auto"` take their natural size, fixed sizes are exact, and `"1fr"`, `"2fr"`, ... share the remaining space. Use `style(align =, valign =)` on the container to position children that do not fill it.

Value

A `Vertical` or `Horizontal` widget.

Examples

```
ui <- vertical(
  label("Title"),
  horizontal(label("left"), label("right"))
)
render_widget(ui, 20, 3)
```

`Widget` *Widget base class*

Description

Every element of a termr interface is a `Widget`. Widgets form a tree: each widget has one parent and an ordered list of children. Most users create widgets with constructor functions such as `label()` or `vertical()`, and define new widget types with `widget()`. Subclassing `Widget` with `R6::R6Class()` is also supported for advanced use.

A subclass usually overrides some of:

- `default_style()`: the built-in `style()` of the widget type;
- `render()`: returns the content as a string, a character vector of lines or `span()`s;
- `on_key(event)`: handle a key press (return `TRUE` when handled);
- `paint(buffer, area, st)`: full control over drawing.

State

Widget state lives in reactive fields: assigning a new value invalidates the widget and schedules a repaint. Several assignments in one event loop tick produce a single repaint.

Public fields

`id` Unique identifier used by `#id` selectors, or `NULL`.

`classes` Character vector of classes used by `.class` selectors.

`help` Optional help text (Markdown) shown on the help screen (F1) while this widget or a descendant has focus.

`paint_states` Names of state fields whose changes only need a repaint (see `invalidate_paint()`); used by `set_state()`.

`focusable` Can the widget receive keyboard focus?

`region` Border box assigned by the layout engine (`region()`).

Active bindings

`parent` The parent widget (read-only).

`children` List of child widgets (read-only).

`app` The App this widget belongs to, or `NULL`.

`type` Widget type name used by type selectors.

`visible` Is the widget shown? Hidden widgets take no space.

`disabled` Disabled widgets cannot be focused or used.

`style` The widget's own `style()` (layered over its defaults).

`border_title` Text shown in the top border (reactive), or `NULL`.

`hovered` Is the mouse pointer over this widget (or a descendant)?

`focused` Does the widget currently have keyboard focus?

Methods

Public methods:

- `Widget$new()`
- `Widget$default_style()`
- `Widget$render()`
- `Widget$mount()`
- `Widget$replace()`
- `Widget$clear()`
- `Widget$remove()`
- `Widget$remove_children()`
- `Widget$walk()`
- `Widget$ancestors()`
- `Widget$query()`
- `Widget$query_one()`
- `Widget$matches()`
- `Widget$on()`
- `Widget$bind()`

- Widget$default_bindings()
- Widget$bindings()
- Widget$bind_reactive()
- Widget$capture_mouse()
- Widget$release_mouse()
- Widget$run_action()
- Widget$post_message()
- Widget$set_timeout()
- Widget$set_interval()
- Widget$animate()
- Widget$run_worker()
- Widget$run_process()
- Widget$focus()
- Widget$blur()
- Widget$add_class()
- Widget$remove_class()
- Widget$toggle_class()
- Widget$has_class()
- Widget$set()
- Widget$get_state()
- Widget$set_state()
- Widget$layout_differs()
- Widget$invalidate()
- Widget$invalidate_paint()
- Widget$refresh()
- Widget$render_lines()
- Widget$content_width()
- Widget$content_height()
- Widget$arrange_children()
- Widget$child_clip()
- Widget$pseudo_states()
- Widget$computed_style()
- Widget$layout_descend()
- Widget$paint()
- Widget$is_enabled()
- Widget$is_displayed()
- Widget$format()
- Widget$print()
- Widget$clone()

`Widget$new()`: Create a widget.

Usage:

```
Widget$new(
  ...,
  id = NULL,
  classes = NULL,
  style = NULL,
  disabled = FALSE,
  visible = TRUE
)
```

Arguments:

`...` Child widgets.

`id` Optional identifier.

`classes` Character vector (or space separated string) of classes.

`style` A `style()`.

`disabled` Is the widget disabled?

`visible` Is the widget visible?

`Widget$default_style()`: The built-in style of this widget type.

Usage:

```
Widget$default_style()
```

`Widget$render()`: Content of the widget (string, lines or spans). Containers return `NULL`.

Usage:

```
Widget$render()
```

`Widget$mount()`: Add child widgets, at the end or at a position. A widget that already has a parent is moved. Ids must be unique within a screen.

Usage:

```
Widget$mount(..., before = NULL, after = NULL)
```

Arguments:

`...` Widgets or lists of widgets.

`before, after` A child widget (or its index) to insert before / after.

`Widget$replace()`: Remove all children and mount new ones.

Usage:

```
Widget$replace(...)
```

Arguments:

`...` Widgets.

`Widget$clear()`: Remove all children (containers). Some widgets redefine `clear()` for their content (e.g. `input()`, `log_view()`).

Usage:

```
Widget$clear()
```

`Widget$remove()`: Detach this widget (and its children) from its parent.

Usage:

```
Widget$remove()
```

`Widget$remove_children()`: Remove all children.

Usage:

```
Widget$remove_children()
```

`Widget$walk()`: This widget followed by all descendants (depth first).

Usage:

```
Widget$walk()
```

`Widget$ancestors()`: Parent, grandparent, ... up to the root.

Usage:

```
Widget$ancestors()
```

`Widget$query()`: Find descendants matching a CSS-like selector.

Usage:

```
Widget$query(selector)
```

Arguments:

`selector` E.g. `"Button"`, `"#name"`, `".danger"`, `"Vertical > Button.primary"`, or several separated by commas.

Returns: A list of widgets.

`Widget$query_one()`: The first descendant matching `selector`.

Usage:

```
Widget$query_one(selector)
```

Arguments:

`selector` A selector, see `query()`.

`Widget$matches()`: Does this widget match `selector`?

Usage:

```
Widget$matches(selector)
```

Arguments:

`selector` A selector, see `query()`.

`Widget$on()`: Handle events of a type that reach this widget (its own events and events bubbling up from descendants).

Usage:

```
Widget$on(type, handler, selector = NULL)
```

Arguments:

`type` Event type, or `"*"` for all.

`handler` `function(event, app)`.

`selector` Optional selector matched against the event sender. The order of `on()` / `app$on()`, `widget$on(type, selector, handler)`, is accepted too.

`Widget$bind()`: Add a key binding to this widget, see `bind()`.

Usage:

```
Widget$bind(key, action = NULL, description = NULL)
```

Arguments:

`key` Key name or a `bind()` binding.

`action` Action name or function.

`description` Optional description.

`Widget$default_bindings()`: Built-in bindings of this widget type.

Usage:

```
Widget$default_bindings()
```

`Widget$bindings()`: All bindings of this widget (type bindings first, so instance bindings win).

Usage:

```
Widget$bindings()
```

`Widget$bind_reactive()`: Keep a field in sync with a reactive function: `fn` runs now and again whenever the `signal()`s it reads change, and its result is assigned to `field` (e.g. `"text"` for a label). The binding ends when the widget is removed.

Usage:

```
Widget$bind_reactive(field, fn)
```

Arguments:

`field` Name of a widget field (an active binding such as `text`).

`fn` A function of no arguments.

`Widget$capture_mouse()`: Receive all mouse events (also outside this widget's region) until `release_mouse()` or the button is released.

Usage:

```
Widget$capture_mouse()
```

`Widget$release_mouse()`: Stop capturing the mouse.

Usage:

```
Widget$release_mouse()
```

`Widget$run_action()`: Run an action, looked up on this widget, its ancestors, then the app.

Usage:

```
Widget$run_action(action)
```

Arguments:

`action` Action name or function.

`Widget$post_message()`: Send a message that bubbles up from this widget to the app. Handlers receive a MessageEvent whose `sender` is this widget.

Usage:

```
Widget$post_message(type, data = list(), bubbles = TRUE)
```

Arguments:

`type` Message name, e.g. `"counter.changed"`.

`data` A list of values.

`bubbles` Should the message bubble to ancestors?

Returns: The event (invisibly), or `NULL` when not attached to an app.

`Widget$set_timeout()`: Call `callback(self, app)` once after `delay` seconds. The timer is cancelled automatically when the widget is removed.

Usage:

```
Widget$set_timeout(delay, callback)
```

Arguments:

`delay` Seconds.

`callback` Function.

Returns: A Timer.

`Widget$set_interval()`: Call `callback(self, app)` every `interval` seconds while the widget is mounted.

Usage:

```
Widget$set_interval(interval, callback)
```

Arguments:

`interval` Seconds.

`callback` Function.

Returns: A Timer.

`Widget$animate()`: Animate a numeric field of this widget. See `animate()`.

Usage:

```
Widget$animate(property, to, ...)
```

Arguments:

`property` Field name.

`to` Target value.

`...` Other arguments of `animate()`.

`Widget$run_worker()`: Run a background worker owned by this widget (cancelled when the widget is removed). See `run_worker()`.

Usage:

```
Widget$run_worker(fn, ...)
```

Arguments:

`fn` Function to run.

`...` Other arguments of `run_worker()`.

`Widget$run_process()`: Run an external program owned by this widget. See `run_process()`.

Usage:

```
Widget$run_process(command, ...)
```

Arguments:

`command` Program to run.

`...` Other arguments of `run_process()`.

`Widget$focus()`: Give keyboard focus to this widget.

Usage:

```
Widget$focus()
```

`Widget$blur()`: Remove keyboard focus from this widget.

Usage:

```
Widget$blur()
```

`Widget$add_class()`: Add classes.

Usage:

```
Widget$add_class(...)
```

Arguments:

`...` Class names.

`Widget$remove_class()`: Remove classes.

Usage:

```
Widget$remove_class(...)
```

Arguments:

`...` Class names.

`Widget$toggle_class()`: Add a class if absent, remove it if present.

Usage:

```
Widget$toggle_class(name)
```

Arguments:

`name` Class name.

`Widget$has_class()`: Does the widget have this class?

Usage:

```
Widget$has_class(name)
```

Arguments:

`name` Class name.

`Widget$set()`: Assign one or more fields, e.g. `app$query_one("#name")$set(value = "")`. R does not allow `app$query_one("#name")$value <- ""`, so use this method when the widget comes from a function call.

Usage:

`Widget$set(...)`

Arguments:

`...` Named values.

Returns: The widget, invisibly (so calls can be chained).

`Widget$get_state()`: Read a state field.

Usage:

```
Widget$get_state(name)
```

Arguments:

`name` Field name.

`Widget$set_state()`: Write a state field; invalidates the widget when the value changes.

Usage:

```
Widget$set_state(name, value)
```

Arguments:

`name` Field name.

`value` New value.

`Widget$layout_differs()`: Would the widget's size-related style differ if it were in the pseudo states `with` rather than `without`? Used to decide whether a focus / hover change needs a layout pass.

Usage:

```
Widget$layout_differs(with, without)
```

Arguments:

`with`, `without` Character vectors of states.

`Widget$invalidate()`: Mark the widget as needing a re-render and schedule a repaint, including a new layout pass (its size may have changed). Repaints are batched per event-loop tick.

Usage:

```
Widget$invalidate()
```

`Widget$invalidate_paint()`: Like `invalidate()` for changes that cannot affect the size or position of anything (a cursor moved, a colour changed): the widget is repainted but the layout is not recomputed.

Usage:

```
Widget$invalidate_paint()
```

`Widget$refresh()`: Force the widget to re-render on the next repaint.

Usage:

```
Widget$refresh()
```

`Widget$render_lines()`: Rendered content as lines of styled segments (cached until the widget is invalidated).

Usage:

```
Widget$render_lines(width = NA_integer_)
```

Arguments:

`width` Available content width (may be `NA` while measuring).

`Widget$content_width()`**:** Natural width of the content area.

Usage:

```
Widget$content_width()
```

`Widget$content_height()`**:** Natural height of the content area for a given width.

Usage:

```
Widget$content_height(width)
```

Arguments:

`width` Content width.

`Widget$arrange_children()`**:** Place the visible children inside the content box. Containers with special behaviour (scrolling, ...) override this.

Usage:

```
Widget$arrange_children(children, inner, st)
```

Arguments:

`children` Visible child widgets.

`inner` Content rectangle of this widget.

`st` Computed style of this widget.

Returns: A list with one rectangle per child.

`Widget$child_clip()`**:** The rectangle children are clipped to when painting.

Usage:

```
Widget$child_clip(st)
```

Arguments:

`st` Computed style of this widget.

`Widget$pseudo_states()`**:** Names of active states used to select state styles.

Usage:

```
Widget$pseudo_states()
```

`Widget$computed_style()`**:** The fully resolved style of the widget.

Usage:

```
Widget$computed_style(inherited = NULL)
```

Arguments:

`inherited` The parent's computed style (computed if `NULL`).

`Widget$layout_descend()`**:** Which children need their own layout? Containers that clip their children (scroll views) skip the ones that are entirely out of view; they get their region but not their descendants.

Usage:

```
Widget$layout_descend(rects)
```

Arguments:

`rects` The rectangles `arrange_children()` returned.

`Widget$paint()`: Paint the widget (not its children) into a buffer.

Usage:

```
Widget$paint(buffer, area, st)
```

Arguments:

`buffer` A ScreenBuffer.

`area` The visible part of `self$region`.

`st` The computed style.

`Widget$is_enabled()`: Is the widget and every ancestor enabled?

Usage:

```
Widget$is_enabled()
```

`Widget$is_displayed()`: Is the widget and every ancestor visible?

Usage:

```
Widget$is_displayed()
```

`Widget$format()`: A one-line description used by `print()`.

Usage:

```
Widget$format(...)
```

Arguments:

`...` Ignored.

`Widget$print()`: Print the widget tree.

Usage:

```
Widget$print(...)
```

Arguments:

`...` Ignored.

`Widget$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`widget` *Define a custom widget type*

Description

`widget()` creates a new widget type and returns its constructor. All behaviour is given as plain functions that receive the widget as their first argument, `self`.

Usage

```
widget(
  name,
  state = list(),
  render = NULL,
  compose = NULL,
  bindings = list(),
  actions = list(),
  on = list(),
  methods = list(),
  style = NULL,
  focusable = FALSE,
  inherit = Widget
)
```

Arguments

`name`	Type name (used by type selectors, e.g. `app$query("Counter")`).
`state`	A named list of state fields. Values created with `reactive()` can have a watcher; plain values are reactive too. Fields are read and assigned as `self$<name>`.
`render`	`function(self)` returning the content: a string, a character vector of lines or `span()`s.
`compose`	`function(self)` returning a list of child widgets.
`bindings`	A list of `bind()` key bindings.
`actions`	A named list of `function(self)`; bindings refer to them by name.
`on`	A named list of event handlers `function(self, event)`, e.g. `list(mount = ..., "button.pressed" = ...)`.
`methods`	A named list of extra methods `function(self, ...)`, callable as `widget$<name>(...)`.
`style`	Default `style()` of the type.
`focusable`	Can widgets of this type receive focus?
`inherit`	Parent class: Widget, Vertical or Horizontal.

Value

A constructor function `function(..., id = NULL, classes = NULL, style = NULL, disabled = FALSE)`. Named arguments in `...` set initial state values; unnamed arguments are child widgets.

Examples

```
Counter <- widget(
  "Counter",
  state = list(count = reactive(0L)),
  render = function(self) sprintf("Count: %d", self$count),
  bindings = list(bind("up", "increment"), bind("down", "decrement")),
  actions = list(
    increment = function(self) self$count <- self$count + 1L,
    decrement = function(self) self$count <- self$count - 1L
  ),
  focusable = TRUE
)
counter <- Counter(count = 5L)
counter$count
render_widget(counter, 12, 1)
```

---

widget_snapshot *Snapshot the safe, structural parts of a widget tree*

Description

Arbitrary widget state and field values are intentionally omitted, so passwords, database credentials, and worker data are not serialized.

Usage

```
widget_snapshot(x)
```

Arguments

x	A widget, Screen, or App.

Value

A deterministic nested list of widget descriptions.

---

widget_snapshot_json *Serialize a structural widget snapshot as JSON*

Description

The JSON form of `widget_snapshot()`. Requires the optional `jsonlite` package.

Usage

```
widget_snapshot_json(x, pretty = TRUE)
```

Arguments

`x`	A widget, Screen, or App.
`pretty`	Format JSON for readability?

Value

A JSON string.

`Worker` *Worker handle*

Description

Returned by `app$run_worker()` and `app$run_process()`. See `run_worker()`.

Public fields

`name` Name of the worker (used in events).

`state` `"running"`, `"completed"`, `"failed"` or `"cancelled"`.

`result` The value returned by the function (workers) or the exit status (processes).

`error` The error message, if it failed.

`timed_out` Did the worker fail because its timeout expired?

`progress` Last reported progress value.

`owner` Widget that owns the worker, or `NULL`.

`stdout, stderr` The most recent output lines (at most 1000).

`kind` `"worker"` (an R function) or `"process"`.

Methods

Public methods:

- `Worker$new()`
- `Worker$cancel()`
- `Worker$is_running()`
- `Worker$pid()`
- `Worker$print()`
- `Worker$clone()`

`Worker$new()`: Create a worker (use `app$run_worker()`).

Usage:

```
Worker$new(name, owner = NULL, callbacks = list())
```

Arguments:

`name` Name.

`owner` Owner widget.

`callbacks` List of callbacks.

`Worker$cancel()`: Stop the worker: an interrupt is sent first (Unix) and the process tree is killed if it has not ended after `grace` seconds.

Usage:

```
Worker$cancel(grace = 0.3)
```

Arguments:

`grace` Seconds to wait before killing.

`Worker$is_running()`: Is the worker still running?

Usage:

```
Worker$is_running()
```

`Worker$pid()`: Process id of the child (or `NA`).

Usage:

```
Worker$pid()
```

`Worker$print()`: Print the worker.

Usage:

```
Worker$print(...)
```

Arguments:

`...` Ignored.

`Worker$clone()`: The objects of this class are cloneable with this method.

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`write_rendered` *Write a static rendering to a UTF-8 file*

Description

Write a static rendering to a UTF-8 file

Usage

```
write_rendered(
  x,
  file,
  format = c("text", "markdown", "html", "svg", "json"),
  width = 80,
  height = 24
)
```

Arguments

`x`	A widget, Screen, App, or ScreenBuffer.
`file`	Output path.
`format`	One of `text`, `markdown`, `html`, `svg`, or `json`.
`width`, `height`	Frame size in cells.

Index

alert_dialog *(modal)*, 81
animate, 4
animate(), *5, 6*, *188*
Animation, 5
App, 6, *17*, *19*, *28*, *49*, *66*, *70*, *105–108*, *111*, *115*, *119, 120*, *155*, *183*, *194, 195*, *197*
app, 16
app(), *6*, *8*, *17*, *84*, *111*

batch *(signals)*, 128
bind, 17
bind(), *9, 10*, *17*, *27*, *187*, *193*
BlurEvent *(Event)*, 49
browse_data, 18
Button, 19
button, 21
button(), *19*

c(), *130*
c.termr_text *(span)*, 130
cell, 22
cell(), *117*
Checkbox, 23
checkbox, 25
checkbox(), *23, 24*
column, 26
column(), *32*, *42*
command, 27
command(), *11*
computed *(signals)*, 128
confirm_dialog *(modal)*, 81
current_app, 28

data_browser *(browse_data)*, 18
data_profile, 41
data_profile(), *28*
data_table, 42
data_table(), *18*, *29*, *32*, *46, 47*, *70*, *97*, *103*, *147*
DataProfile, 28
DataTable, 29
db_connection, 44
db_connection(), *44–46*, *139*
db_explorer, 44
db_metadata, 45
db_query_source, 45
db_query_source(), *139*
db_table_source, 46
diff_screen, 47
diff_screen(), *89*, *115*
dispose *(signals)*, 128
dropdown, 48
dropdown(), *126*

Event, *11*, 49, *50–55*

FocusEvent *(Event)*, 49

Grid, 56
grid_layout, 57
grid_layout(), *56*

Horizontal, 58, *193*
horizontal *(vertical)*, 181
horizontal(), *58*

identical(), *104*
Input, 59
input, 64
input(), *59*, *185*
inspect_signal, 66
inspect_widget, 66

json_view, 67

key_event, 69
key_value, 69
key_value(), *68*, *70*, *97*, *103*
KeyEvent *(Event)*, 49
KeyValue, 68

knit_termr, 70

Label, 71, *171*
label, 72
label(), *71*, *130*, *182*
log_view, 75
log_view(), *73*, *185*
LogView, 73

markdown_view, 77
markdown_view(), *76*
MarkdownView, 76
MessageEvent, *187*
MessageEvent *(*Event*)*, 49
Metric, 78
metric, 80
metric(), *78*
modal, 81
modal(), *13*, *82*, *83*
ModalScreen, *13*, 82
motion_reduced, 84
motion_reduced(), *7*, *17*
MountEvent *(*Event*)*, 49
MouseEvent *(*Event*)*, 49

on, 84
on(), *7*, *9*, *10*, *17*, *187*
option_list, 87
option_list(), *85*
OptionList, 85

Panel, 87
panel, 88
panel(), *87*
PasteEvent *(*Event*)*, 49
patch_to_ansi, 89
peek *(*signals*)*, 128
process_view, 91
process_view(), *89*, *90*
ProcessView, 89
progress_bar, 94
progress_bar(), *4*, *92*, *93*
ProgressBar, 92
property_grid, 96
property_grid(), *70*, *95–97*, *102*, *103*
PropertyGrid, 95

R6::R6Class(), *182*
r_highlighter, 114
radio_button *(*radio_set*)*, 101
radio_button(), *98*
radio_set, 101
radio_set(), *98–100*
RadioButton, 98
RadioSet, 99
reactive, 102
reactive(), *129*, *193*
record_view, 102
record_view(), *70*, *97*, *103*
region, 104
region(), *105*, *117*, *118*, *183*
register_layout, 104
render_html, 105
render_lines *(*render_text*)*, 107
render_markdown, 106
render_svg, 106
render_text, 107
render_widget, 108
ResizeEvent *(*Event*)*, 49
Rule, 108
rule, 110
rule(), *108*, *109*
run, 110
run(), *6*, *17*
run_example, 111
run_process *(*run_worker*)*, 112
run_process(), *14*, *189*
run_worker, 112
run_worker(), *13*, *14*, *188*, *195*

Screen, *6*, *7*, *13*, *70*, *82*, *105–108*, 115, *119*, *120*, *194*, *195*, *197*
screen_buffer, 119
screen_buffer(), *115*
screen_snapshot, 119
screen_snapshot(), *120*
screen_snapshot_json, 120
ScreenBuffer, *7*, *40*, *47*, *70*, *105–109*, 115, *119*, *120*, *124*, *141*, *152*, *167*, *178*, *192*, *197*
scroll_view, 124
scroll_view(), *4*, *78*, *120*, *121*
ScrollView, *95*, 120
Select, 126
set_interval *(*set_timeout*)*, 128
set_interval(), *169*
set_timeout, 128
set_timeout(), *169*, *170*

signal (signals), 128
signal(), *20*, *72*, *142*, *187*
signals, 128
span, 130
span(), *71*, *72*, *182*, *193*
Sparkline, 131
sparkline, 132
sparkline(), *131*
Spinner, 133
spinner, 135
spinner(), *133*, *134*
split_pane, 138
split_pane(), *136*, *137*
SplitPane, 136
sql_editor, 139
sql_highlighter, 140
status_bar, 142
status_bar(), *140*, *141*
StatusBar, 140
strip_ansi, 143
style, 143
style(), *22*, *23*, *25*, *41–43*, *48*, *57*, *65*, *67*, *69*, *72*, *75*, *77*, *80*, *87*, *88*, *95*, *97*, *101*, *103*, *104*, *110*, *117*, *125*, *130*, *133*, *135*, *138*, *142*, *146*, *153*, *168*, *179*, *182*, *183*, *185*, *193*
stylesheet, 145
stylesheet(), *9*, *13*, *17*
stylesheet_file (stylesheet), 145

tab (tabs), 152
tab(), *148*, *150*, *151*
table_filter, 146
table_filter(), *46*
table_source, 147
table_source(), *32*, *33*, *45*, *46*
TabPane, 148, *149*
Tabs, 149
tabs, 152
tabs(), *149*, *150*
terminal_capabilities, 153
termr_theme, 154
termr_theme(), *7*, *9*, *17*
test_app, 155
test_app(), *69*
testthat::expect_snapshot(), *156*
text_area, 168
text_area(), *114*, *139*, *140*, *156*
TextArea, *139*, 156
Timer, *128*, 169, *188*
Toast, 171
tree_node, 178
tree_node(), *172*, *175*, *179*
tree_view, 179
tree_view(), *67*, *172*, *175*, *176*
TreeNode, 172, *176*
TreeView, 175

UnmountEvent (Event), 49
unregister_layout, 180
untracked (signals), 128
update_signal (signals), 128

Vertical, *28*, *87*, *89*, *99*, *148*, 181, *193*
vertical, 181
vertical(), *181*, *182*

watch (signals), 128
Widget, *19*, *20*, *23*, *24*, *28*, *29*, *33*, *56*, *58–60*, *68*, *71*, *73*, *74*, *76*, *78*, *79*, *82*, *83*, *85–87*, *89*, *90*, *92*, *93*, *95*, *98–100*, *108*, *109*, *115*, *120*, *122*, *126*, *127*, *131–134*, *136*, *137*, *140*, *141*, *148–150*, *156*, *160*, *171*, *175*, *176*, *181*, 182, *193*
widget, 193
widget(), *17*, *102*, *182*
widget_snapshot, 194
widget_snapshot(), *194*
widget_snapshot_json, 194
Worker, *89*, *114*, 195
write_rendered, 196