Package 'ggpaintr'

July 7, 2026

Type Package

Title Build Formula-Driven 'shiny' Apps for 'ggplot2'

Version 0.11.1

Description Turns a single 'ggplot2'-style formula string into a small 'shiny' application. Placeholder tokens in the formula become input widgets automatically; the package completes the expression with the current input values, renders the plot, shows the generated code, and supports uploaded datasets in different formats.

License GPL-3

URL https://willju-wangqian.github.io/ggpaintr/, https://github.com/willju-wangqian/ggpaintr

BugReports https://github.com/willju-wangqian/ggpaintr/issues

Depends R (>= 4.1.0), ggplot2 (>= 3.3.0)

Imports assertthat (>= 0.2.0), cli (>= 2.2.0), htmltools, rlang (>= 1.0.0), shiny (>= 1.6.0), shinyWidgets (>= 0.6.4)

Suggests bslib (>= 0.5.0), dplyr, ellmer, jsonlite, knitr, miniUI, pkgdown, plotly, purrr, readxl, rmarkdown, rstudioapi, shinytest2, testthat (>= 3.1.7), withr, writexl, xaringanExtra

VignetteBuilder knitr

Encoding UTF-8

RoxygenNote 7.3.3

Config/testthat/edition 3

Config/testthat/start-first e2e-vignette-examples-shinytest2, j12-spec-and-renderui-emission-journey, shared-source-panel-multi-instance, adr0024-companion-entry-point, prologue-reader-fn-mirror, j2-prologue-csv-upload-journey, j11-app-basic-journey, default-seeding-with-source-upstream, adr15-consumer-binding

NeedsCompilation no

Author Wangqian Ju [aut, cre] (ORCID: <https://orcid.org/0000-0002-9977-377X>),
Jinji Pang [aut] (ORCID: <https://orcid.org/0000-0002-2267-2313>),
Zhili Qiao [aut] (ORCID: <https://orcid.org/0000-0002-8154-0898>)

Maintainer Wangqian Ju <wju@iastate.edu>

Repository CRAN

Date/Publication 2026-07-07 10:10:02 UTC

Contents

`embellish_helpers` *Built-in helpers for a placeholder's plain-R evaluation slot*

Description

A placeholder-embellished ggplot expression must stay valid plain R that renders the original plot with no app running. Each placeholder carries a callable (the `embellish_eval =` argument of the `ptr_define_placeholder_*()` constructors) that supplies its plain-R meaning when the naked expression is evaluated outside `ptr_app()`. These two factories are the built-in callables for that slot:

Usage

```
embellish_identity()

embellish_symbol_to_string()
```

Details

- `embellish_identity()` returns the identity `function(x, ...) x` — the slot's default behaviour. The placeholder call becomes a no-op wrapper: it returns its argument unchanged.
- `embellish_symbol_to_string()` returns a function that captures its argument *unevaluated* and turns column references into a character vector of names. This is the pattern a column-selecting consumer needs so the naked expression works inside a tidyselect verb: tidyselect evaluates an unknown wrapper call in non-masked scope, where bare column symbols throw "object not found"; returning the names as strings sidesteps that because tidyselect accepts selection by name.

These helpers are *author-controlled* plain-R semantics, never derived — only the author knows the intended live-R meaning of a placeholder.

Value

Each factory returns a function of signature `function(x, ...)`. `embellish_identity()`'s function returns its first argument `x` unchanged. `embellish_symbol_to_string()`'s function returns a character vector of column names captured from the unevaluated `x`.

Examples

```
f <- embellish_identity()
f(5L)

g <- embellish_symbol_to_string()
g(c(mpg, hp))
g(mpg)
```

`ppLayerOff` *Off-by-default layer wrapper*

Description

ADR-0020 *structural* keyword that marks a layer as off-by-default in `ptr_app()`. Inside `ptr_app()` / `ptr_server()` the parser sees the wrapper and unwraps it at translate time, stamping the boot-state metadata on the resulting node: `ppLayerOff(layer_expr, hide = TRUE)` becomes a `ptr_layer` with `default_active = FALSE`. The wrapper itself never appears in the typed tree.

Usage

```
ppLayerOff(layer_expr, hide = TRUE)
```

Arguments

`layer_expr`	A ggplot2 layer expression (e.g. `geom_point()`, `facet_wrap(~ cyl)`). Evaluated only when `hide = FALSE`.
`hide`	A length-1 logical literal (`TRUE` or `FALSE`). In `ptr_app()` formulas this MUST be a literal — the translator aborts on a non-literal so the formula remains the single source of truth for the app's boot state. Defaults to `TRUE`.

Details

Outside `ptr_app()` it behaves per its R semantics so naked-ggplot scripts still render: `ppLayerOff(geom_point(), TRUE)` returns `NULL` (so `ggplot(mtcars, aes(x = mpg, y = wt)) + ppLayerOff(geom_point(), TRUE)` renders without the hidden layer); `ppLayerOff(geom_point(), FALSE)` returns the layer.

For the pipeline-stage sibling that exposes a user-toggleable checkbox (ADR-0021), see `ppVerbSwitch()`.

Value

Outside `ptr_app()`: `NULL` when `hide = TRUE`, otherwise the evaluated `layer_expr`.

Examples

```
library(ggplot2)
# Naked-R semantics: hide = TRUE drops the layer to NULL.
p1 <- ggplot(mtcars, aes(x = mpg, y = wt)) +
  ppLayerOff(geom_point(), TRUE)        # equivalent to no geom_point
p2 <- ggplot(mtcars, aes(x = mpg, y = wt)) +
  ppLayerOff(geom_point(), FALSE)       # the layer is added

# Inside ptr_app(), the wrapper becomes a node-level default and a
# boot-state-off checkbox:
if (interactive()) {
  ptr_app(ggplot() + ppLayerOff(geom_point(aes(x = mpg, y = wt)), TRUE))
}
```

---

ppVar *Placeholder Identity Helpers*

Description

These are the plain-R callables returned by registering the five built-in ggpaintr placeholders (`ppVar`, `ppNum`, `ppText`, `ppExpr`, `ppUpload`). Inside a formula passed to `ptr_app()` / `ptr_server()` the parser recognises calls to these names as placeholder invocations and binds them to Shiny widgets (see `vignette("ggpaintr-tutorial")`). Outside `ptr_app()` they behave as plain R functions: the first four return their argument unchanged (identity), so a formula such as `aes(x = ppVar(mpg))` evaluates identically to `aes(x = mpg)` under ggplot2's tidy-eval. `ppUpload` is identical when called with an argument (`ppUpload(penguins)` returns `penguins`), so a formula such as `ppUpload(penguins) |> filter(...) |> ggplot(...)` evaluates as plain R when `penguins` is in scope. The no-arg form `ppUpload()` aborts outside `ptr_app()` — it is meaningful only as a placeholder slot.

Usage

```
ppVar(x = NULL, ...)

ppNum(x = NULL, ...)

ppText(x = NULL, ...)

ppExpr(x = NULL, ...)

ppUpload(x, ...)
```

Arguments

`x`	A column name (`ppVar`), numeric (`ppNum`), string (`ppText`), expression (`ppExpr`), or dataset name/value (`ppUpload`). Passed through unchanged.
`...`	Additional arguments (e.g. named arguments consumed by a custom placeholder's `named_args` schema). Ignored by the built-in identity implementation.

Value

The input value unchanged. The no-arg form `ppUpload()` does not return; it aborts with a guard message.

Examples

```
# Identity inside ggplot2's tidy-eval:
library(ggplot2)
p1 <- ggplot(mtcars, aes(x = mpg)) + geom_histogram(bins = 10)
p2 <- ggplot(mtcars, aes(x = ppVar(mpg))) + geom_histogram(bins = 10)
# p1 and p2 produce the same plot.

# Inside ptr_app() / ptr_server(), the same call binds to a column picker:
if (interactive()) {
  ptr_app(ggplot(mtcars, aes(x = ppVar(mpg), y = ppVar(wt))) + geom_point())
}
```

---

ppVerbSwitch *Switchable pipeline-stage wrapper*

Description

ADR-0021 *structural* keyword that marks a pipeline stage as user- toggleable in `ptr_app()`. The boolean argument is `switch_on` (positive sense: TRUE applies the verb, FALSE skips it) and an optional `label` carries the UI text for the resulting checkbox. Inside `ptr_app()` / `ptr_server()` the parser sees the wrapper and unwraps it at translate time, stamping the boot-state metadata + UI label onto the resulting node. The wrapper itself never appears in the typed tree.

Usage

```
ppVerbSwitch(.data, verb_expr, switch_on = TRUE, label = NULL)
```

Arguments

`.data`	A data frame or pipe-supplied dataset (the implicit `.data` slot when used as a pipeline stage).
`verb_expr`	A data-pipeline verb call (e.g. `mutate(mpg = mpg + 100)`, `filter(cyl == 6)`). Evaluated with `.data` inserted as the first positional argument only when `switch_on = TRUE`.
`switch_on`	A length-1 non-NA logical literal. In `ptr_app()` formulas this MUST be a literal — the translator aborts on a non-literal so the formula remains the single source of truth for the app's boot state. Defaults to `TRUE` (apply the verb).
`label`	Optional length-1 character used as the checkbox label inside `ptr_app()`. Ignored by the naked-R path. Defaults to `NULL`.

Details

Outside `ptr_app()` it behaves per its R semantics so naked-dplyr scripts still render: `ppVerbSwitch(.data, mutate(x = 1), FALSE)` returns `.data` unchanged; `ppVerbSwitch(.data, mutate(x = 1), TRUE)` routes `.data` through the verb call. `label` is metadata-only outside `ptr_app()` (the naked-R path ignores it).

Value

Outside `ptr_app()`: returns `.data` unchanged when `switch_on = FALSE`, otherwise the result of `verb_expr` applied to `.data`.

Data-argument position

`ppVerbSwitch(.data, verb_expr, switch_on = TRUE)` inserts `.data` as the **first positional argument** of `verb_expr` when `switch_on` is TRUE. This matches the tidyverse convention and the translator's pipeline-stage handling; non-tidyverse verbs that take data in a later argument are not supported (use a lambda stage or a named wrapper instead).

Examples

```
if (requireNamespace("dplyr", quietly = TRUE)) {
  # Naked-R semantics: switch_on = FALSE leaves the data unchanged.
  identical(
    ppVerbSwitch(mtcars, dplyr::mutate(mpg = mpg + 100), FALSE),
    mtcars
  )

  # switch_on = TRUE routes .data through the verb.
  result <- ppVerbSwitch(mtcars, dplyr::filter(mpg > 20), TRUE)
  nrow(result)  # 14
}

# Inside ptr_app(), the wrapper becomes a node-level default + a
# labelled boot-state-on checkbox:
if (interactive()) {
  ptr_app(
    "mtcars |> ppVerbSwitch(dplyr::filter(mpg > 20), TRUE, label = 'Filter') |>
     ggplot(aes(x = mpg, y = wt)) + geom_point()"
  )
}
```

---

ptr_app *Build a Shiny App from a* ggpaintr *Formula*

Description

Translates `formula` into the typed AST, builds the per-layer panel UI, and wires the server end-to-end. Returns a `shiny.appobj` ready to be run. This page is the canonical reference for the formula grammar and the empty-call cleanup rule used by every public entry point.

Usage

```
ptr_app(
  formula,
  envir = parent.frame(),
  ui_text = NULL,
  expr_check = TRUE,
  safe_to_remove = character(),
  css = NULL,
  spec = NULL
)
```

Arguments

formula	Either a single character scalar containing a ggplot expression with ggpaintr placeholders, or an unquoted ggplot expression supplied directly. Expression-mode is captured with rlang::enexpr() at the public boundary, then deparsed to a string before reaching the shared translate pipeline; both modes produce equivalent apps. A bare symbol bound to a string in the calling frame (e.g. f <- "..."; ptr_app(f)) is resolved and treated as string mode. Pre-quoted wrappers (rlang::expr(), rlang::quo(), base::quote(), base::bquote()) at the captured root are unwrapped one level. !! splicing inside the captured expression is honoured via rlang::enexpr(). **Native pipe (|>) caveat:** in expression mode, R's parser desugars |> before capture, so the rendered code shows the desugared nested-call form. Stay in string mode (or use %>%) if you need |> preserved.
envir	Environment used to resolve local data objects.
ui_text	Optional named list of copy overrides; see ptr_ui_text() for the full schema and current defaults.
expr_check	Controls formula-level ppExpr validation. Three modes: TRUE (default) applies the built-in denylist + AST walker; FALSE disables formula-level validation (for local prototyping with trusted formulas only); a list with deny_list and/or allow_list entries (character vectors) customises the formula-level policy without disabling it. Code typed into a ppExpr box at runtime is always screened against the built-in denylist, regardless of this setting. For the walker model see the safety chapter of the ggpaintr book (development-version docs): https://willju-wangqian.github.io/ggpaintr-book/safety.html.
safe_to_remove	Character vector of additional function names whose zero-argument calls should be dropped after placeholder substitution leaves them empty. Defaults to character(). See **Empty-call cleanup** below. A user-typed ppExpr always wins — whatever the user enters into an ppExpr box is honoured verbatim, even if its top-level name is in safe_to_remove.
css	Optional character vector of paths to additional CSS files. Each is served as a static resource and linked after ggpaintr's bundled stylesheet, so its rules override the default .ptr-* styling. Relative url(...) references inside a file resolve against that file's own directory. Defaults to NULL (no extra stylesheet).
spec	An optional named list of fully-qualified Shiny input id -> value, used to override widget defaults at session boot.

Value

A `shiny.appobj`.

Formula grammar

A `ggpaintr` formula is a single `ggplot()` call written as a string. Drop one of five placeholder keywords anywhere a value would normally go, and the runtime substitutes the user's input back into the expression at render time.

`ppVar` Column picker, data-aware. Renders as a `selectInput` populated with the upstream data's column-name vector. Example: `aes(x = ppVar)`.

`ppText` Free-text input. Renders as a `textInput`. Example: `labs(title = text)`.

`ppNum` Numeric input. Renders as a `numericInput`. Example: `geom_point(size = ppNum)`.

`ppExpr` Code editor, validated by `expr_check`. The only keyword that accepts arbitrary R code; for the safety model see the ggpaintr book's safety chapter (development-version docs, https://willju-wangqian.github.io/ggpaintr-book/safety.html). Example: `facet_wrap(ppExpr)`.

`ppUpload` File picker, returns a data frame. Renders as a `fileInput` plus an optional dataset-name textbox. Accepted formats: `.csv`, `.tsv`, `.rds`, `.xlsx`, `.xls`, `.json`. Uploaded data is normalized via `ptr_normalize_column_names()` automatically. Example: `ggplot(ppUpload, ...)`.

Any keyword occurrence may carry `shared = "<id>"` to lift the widget out of its per-layer panel into a top-level shared section. Used by `ptr_app_grid()` to drive multiple plots from one control. See `vignette("ggpaintr-tutorial")` for worked examples of each keyword.

Empty-call cleanup

When a placeholder resolves to "missing" (an empty `ppVar` pick, a blank `ppText`, a cleared `ppNum`, an unchecked layer checkbox), its argument is dropped from the generated code. If the surrounding call is left empty and its bare name is in the curated cleanup list, the whole call disappears too. This rule applies to both placeholder-driven empties and user-authored literal empty calls like `+ labs()`.

Curated `ggplot2` names that are dropped when empty:

```
theme, labs, xlab, ylab, ggtitle,
facet_wrap, facet_grid, facet_null,
xlim, ylim, lims, expand_limits,
guides, annotate, annotation_custom,
annotation_map, annotation_raster,
aes, aes_, aes_q, aes_string,
element_text, element_line, element_rect,
element_point, element_polygon, element_geom
```

Empty calls to similar no-op helpers from `dplyr`, `tidyr`, `tibble`, `pillar`, `purrr`, `stringr`, `forcats`, `lubridate`, and `hms` are covered by the same rule.
`geom_*()` and `stat_*()` layers are **never** dropped, regardless of whether they end up empty — they inherit their aesthetics from `ggplot()` and remain meaningful with no arguments.

`element_blank()` is intentionally **not** in the cleanup list: its empty form is a meaningful "suppress" directive, not a no-op.

Third-party helpers (e.g. `pcp_theme()` from ggpcp) are not in the cleanup list — being absent is the "removal safety unknown" signal. Use `safe_to_remove = c("pcp_theme")` to opt a specific name in.

See Also

`ptr_app_bslib()` for the same contract with a `bslib` theme; `ptr_app_grid()` for multi-plot apps with shared widgets; `ptr_define_placeholder_value()` et al. for registering custom keywords; `ptr_ui_text()` for copy overrides; `ptr_css()` for the `css =` argument and themable CSS custom properties; `vignette("ggpaintr-tutorial")` for tutorial examples.

Examples

```
if (interactive()) {
  # Expression mode (primary): pass the unquoted ggplot expression.
  ptr_app(ggplot(mtcars, aes(x = ppVar, y = ppVar)) + geom_point())
  # `!!` splices a value into the expression.
  col <- rlang::sym("mpg")
  ptr_app(ggplot(mtcars, aes(x = !!col, y = ppVar)) + geom_point())
  # String mode (fallback): the same formula as text.
  ptr_app("ggplot(mtcars, aes(x = ppVar, y = ppVar)) + geom_point()")
}
```

---

ptr_arg_validators *Argument validators for placeholder definitions (`ptr_arg_*`)*

Description

These helpers are factories that return a closure of shape `function(arg_expr) -> canonical_value | abort()`. The closure validates the unevaluated R expression captured as a placeholder's positional default argument and returns a canonical value, or aborts with a clear message.

Usage

```
ptr_arg_symbol_or_string(vector = FALSE)

ptr_arg_string(vector = FALSE)

ptr_arg_symbol(vector = FALSE)

ptr_arg_numeric(vector = FALSE, length = NULL)

ptr_arg_expression()
```

Arguments

vector	Logical scalar (default FALSE). When TRUE, the validator parses a c(...) literal element-by-element and returns the whole vector instead of a single scalar.
length	Optional integer length required of the resulting numeric vector; honored only when vector = TRUE. NULL (the default) imposes no length check.

Details

The validators operate on AST only: they do not call eval(), parse(), or any deparse-and-reparse cycle on their input. The numeric helper ptr_arg_numeric() (scalar by default, vector via vector = TRUE) walks the AST against the constant-fold allowlist registry (see ptr_register_constant_fold()) and then evaluate in a sealed environment whose only bindings are the registered names.

Symbol policy is per-helper:

- ptr_arg_symbol_or_string() accepts a bareword symbol (returned as its character name, preserving non-syntactic / backticked names) or any single string literal (including the empty string).
- ptr_arg_symbol() accepts only a bareword symbol (returned as its character name); rejects string literals, numbers, and compound calls.
- ptr_arg_string() accepts only a single string literal (including the empty string); rejects symbols and numbers.
- ptr_arg_numeric() accepts any AST whose every node is a syntactic literal or a registered constant-fold name; in the default scalar mode the result must be a length-one non-NA numeric. For the vector form use ptr_arg_numeric(vector = TRUE, length = NULL).
- ptr_arg_expression() is a verbatim store: it returns its input unchanged so it can later be evaluated in the data context. As a convenience it emits a one-shot warning if the user wraps the expression in quote(), bquote(), rlang::ppExpr(), or rlang::quo() (the wrapper is stored verbatim).

Each of ptr_arg_symbol_or_string(), ptr_arg_symbol(), ptr_arg_string(), and ptr_arg_numeric() takes a vector flag. With vector = FALSE (the default) the validator parses a single scalar element and returns a length-one value. With vector = TRUE it parses a c(...) literal element-by-element (each element subject to the helper's scalar element rule) and returns the whole vector; a lone element is treated as a length-one vector. For ptr_arg_numeric(vector = TRUE) the optional length check (honored only in vector mode) asserts the parsed vector's length.

Value

A closure that takes an unevaluated expression and returns the canonical default value, or aborts.

Examples

```
is_symbol_ok <- ptr_arg_symbol_or_string()
is_symbol_ok(quote(mpg))
is_symbol_ok("mpg")

is_num <- ptr_arg_numeric()
is_num(5)
```

```
is_num(quote(2 * pi))

is_vec <- ptr_arg_numeric(vector = TRUE, length = 2L)
is_vec(quote(c(0, 1)))
```

`ptr_clear_placeholder` *Remove user-registered placeholders*

Description

Unregisters placeholders added with `ptr_define_placeholder_value()`, `ptr_define_placeholder_consumer()`, or `ptr_define_placeholder_source()`. The five built-in placeholders (`ppVar`, `ppText`, `ppNum`, `ppExpr`, `ppUpload`) and the two structural keywords (`ppLayerOff`, `ppVerbSwitch`) are never removed.

Usage

```
ptr_clear_placeholder(keyword = NULL)
```

Arguments

`keyword`	Optional single string. When supplied, only that placeholder is removed. When omitted (the default), every user-registered placeholder is removed.

Value

The character vector of keywords that were removed, invisibly.

Examples

```
ptr_define_placeholder_value(
  "demo_kw",
  build_ui = function(node, ...) shiny::textInput(node$id, "demo"),
  resolve_expr = function(value, node, ...) value
)
ptr_clear_placeholder("demo_kw")
```

ptr_constant_fold_registry *Constant-fold allowlist registry*

Description

The numeric default-argument validator `ptr_arg_numeric()` (scalar by default, vector via `vector = TRUE`) walks the placeholder's default-argument AST against an allowlist of function and constant names. Authors can extend the allowlist with `ptr_register_constant_fold()` when their placeholder definitions need additional pure operators.

Usage

```
ptr_register_constant_fold(name, value)

ptr_clear_constant_fold(name = NULL)

ptr_constant_fold_keywords()
```

Arguments

`name`	Character scalar function or constant name.
`value`	Function or numeric constant to bind under `name`.

Details

Built-in entries seeded at package load:

- Arithmetic: `-`, `+`, `*`, `/`, `^`, `%%`, `%/%`
- Sequence constructors: `:`, `c`, `seq`, `seq.int`, `seq_len`, `seq_along`

Syntactic literals (`TRUE`, `FALSE`, `NA`, `NA_integer_`, `NA_real_`, `Inf`, `NaN`) are recognised by the walker directly and never need registration. `pi` resolves through the registry's parent (`baseenv()`).

Value

`ptr_register_constant_fold()` and `ptr_clear_constant_fold()` return `invisible(NULL)`. `ptr_constant_fold_keywords()` returns a character vector of currently registered names.

Examples

```
ptr_register_constant_fold("log10", log10)
ptr_arg_numeric()(quote(log10(100)))
ptr_clear_constant_fold("log10")
```

`ptr_define_placeholder_consumer` *Define a data-consumer placeholder (e.g. column picker)*

Description

A *consumer* placeholder is a value placeholder that additionally receives the columns of the upstream data frame — typically a column picker. The built-in example is `ppVar`. See `vignette("ggpaintr-tutorial")` § "Defining your own placeholders" (consumer role).

Usage

```
ptr_define_placeholder_consumer(
  keyword,
  build_ui,
  resolve_expr,
  validate_session_input = NULL,
  parse_positional_arg = NULL,
  parse_named_args = list(),
  embellish_eval = NULL,
  ui_text_defaults = list(label = "Pick a column for {param}")
)
```

Arguments

`keyword`, `ui_text_defaults`
: See `ptr_define_placeholder_value()`.

`build_ui`
: `function(node, cols, data, label = NULL, selected = NULL, ...)` returning a Shiny tag. `cols` is a character vector of upstream column names (use as `choices`); `character(0)` before upstream resolves. `data` is the upstream data frame, or `NULL` while pending — read it only when you need column types / levels / ranges.

  Widget-seeding contract — `selected`. Same precedence, omit-on-no-default, render-empty-when-empty, and never-read- `node$default` rules as for value placeholders — see the "Widget-seeding contract" block on `ptr_define_placeholder_value()` for the authoritative description.

  Consumer-specific rule: filter through `intersect()`. Always pass `selected = intersect(selected %||% character(0), cols)` (or the equivalent column filter) to the underlying widget, not bare `selected`. Three cases collapse into one line:

  - User-cleared widget (`selected` is empty) → `intersect()` returns `character(0)`, widget renders empty. Honors the render-empty-when-empty rule.
  - Stale pick after upstream data swap (`selected` names a column no longer in `cols`) → `intersect()` drops it cleanly. Without this, `selectInput` silently falls back to its first choice (silent data mutation) and `shinyWidgets::pickerInput` enters a similarly broken state.

- Valid pick → `intersect()` is a no-op, value flows through.

`resolve_expr` `function(value, node, ...)`. For a column picker the typical body is `rlang::sym(value)` so the bare column name is spliced as an identifier rather than a string literal. See `ptr_define_placeholder_value()` for allowed return types and the `NULL`-prunes-the-argument convention.

`validate_session_input`
Optional `function(value, ctx)` called before `resolve_expr`. Return `TRUE` / `NULL` to accept; return a single character string to reject (surfaced inline as the error message, layer pruned). Useful when a stale selection no longer matches any upstream column after a data swap, or when only certain column types are admissible. `ctx` is a plain list with named fields: `node` (the placeholder AST node, carries `$id`, `$keyword`, `$args`), `keyword` (convenience alias for `node$keyword`), `upstream_cols` (character vector of upstream column names — the same value `build_ui` received as `cols`), and `data` (the upstream data frame — the same object `build_ui` received as `data`). `ctx$upstream_cols` and `ctx$data` may both be `NULL` while upstream resolution is pending; the validator is not invoked when upstream has not yet resolved (the substitute walker skips the hook in that case). ggpaintr invokes this function as `validate_input(value, ctx)` — no other positional or named arguments are passed, and `ctx` carries exactly the four fields above. The signature does not require `...`.

`parse_positional_arg`, `parse_named_args`
See `ptr_define_placeholder_value()`. Consumer placeholders use the same arg-schema slots; the `ppVar` built-in passes a column-name validator here when used as `ppVar(mpg)`.

`embellish_eval` Optional `function(x, ...)` body used when the placeholder is called as a plain-R function. `NULL` (default) supplies the identity from `embellish_identity()` (`function(x, ...) x`), matching the legacy `ppVar`-style `aes()` NSE shape (the symbol-passthrough convention). Override to give the consumer a non-identity plain-R meaning (e.g. `embellish_symbol_to_string()`).

Value

The runtime callable (identity by default; override with `embellish_eval = ...`). Also called for its registration side effect; use `ptr_clear_placeholder()` to remove it.

See Also

`vignette("ggpaintr-tutorial")` § "Defining your own placeholders"; `ptr_define_placeholder_value()`, `ptr_define_placeholder_source()`.

Examples

```
# A consumer that picks a numeric-only column.
# Note: `selected = NULL` formal (per the seeding contract) plus
# intersect() filter (per the consumer-specific rule). Empty input
# renders empty; a stale pick after a data swap drops cleanly.
ptr_define_placeholder_consumer(
  keyword = "numvar",
  build_ui = function(node, cols, data, label = NULL,
```

```
                    selected = NULL, ...) {
    numeric_cols <- if (is.null(data)) character(0) else
      names(data)[vapply(data, is.numeric, logical(1))]
    retained <- intersect(selected %||% character(0), numeric_cols)
    shiny::selectInput(node$id, label = label %||% "Numeric column",
                       choices = numeric_cols, selected = retained)
  },
  resolve_expr = function(value, node, ...) {
    if (length(value) != 1L || !nzchar(value)) return(NULL)
    rlang::sym(value)
  },
  validate_session_input = function(value, ctx) {
    if (length(value) == 1L && value %in% ctx$upstream_cols) TRUE
    else "Pick a column that exists in the upstream data."
  }
)
ptr_clear_placeholder("numvar")
```

`ptr_define_placeholder_source`
Define a data-source placeholder (e.g. upload, database table)

Description

A *source* placeholder produces a data frame the rest of the formula reads from. Built-in example: `ppUpload`. Custom examples: database tables, built-in datasets, URL fetches.

Usage

```
ptr_define_placeholder_source(
  keyword,
  build_ui,
  resolve_data,
  resolve_expr = NULL,
  shortcut = FALSE,
  parse_positional_arg = NULL,
  parse_named_args = list(),
  embellish_eval = NULL,
  ui_text_defaults = list(label = "Provide a data source for {param}")
)
```

Arguments

`keyword`, `ui_text_defaults`
: See `ptr_define_placeholder_value()`. See `vignette("ggpaintr-tutorial")` § "Defining your own placeholders" (source role).

`build_ui` `function(node, label, ...)` returning a Shiny tag — same shape as in `ptr_define_placeholder_value()`. Render ONLY the source's data-payload widget at `inputId = node$id` (e.g. a `fileInput`, a `selectInput` chooser). With `shortcut = TRUE` the framework emits the sibling shortcut `textInput` (at `node$shortcut_id`) for you (ADR 0025 item #7) — do NOT render it yourself, or the id would be bound twice. A source whose only entry point is the shortcut (e.g. an env-name loader) may return `NULL`.

Seeding — same opt-in shape as the other two helpers: declare an optional `selected = NULL` formal (or accept `...`) to receive the seeded value. Seeding is **boot-only**: a `spec=` entry naming this source wins on the first render, and on every later render (e.g. a tree rebuild) the user's current pick is authoritative — the spec seed never snaps the widget back. The built-in `ppUpload` declines this because a Shiny `fileInput()` can never be seeded programmatically; custom sources whose primary widget *can* be seeded (e.g. a `selectInput` dataset chooser) should accept it.

`resolve_data` `function(value, node, ...)` returning a `data.frame` (the data downstream consumers read from), or `NULL` to signal "no data yet". Throw via `rlang::abort()` for malformed inputs.

`resolve_expr` Optional. `function(value, node, ...)` returning the expression spliced into the rendered code at the placeholder's position — i.e. *how the data is referred to* in the reproducible call, not the data itself. Default `rlang::sym(value)` works when the widget's value is already the symbol you want. Override to make the rendered code re-fetch instead of referencing an in-session object, e.g. `function(value, node, ...) rlang::ppExpr(read.csv(!!value$datapath))`. With `shortcut = TRUE`, `value` here is the *shortcut* input's value (e.g. the typed dataset name), **not** the primary payload — the built-in `ppUpload` relies on this so the default splices the typed name as a bare symbol.

`shortcut` Single logical (default `FALSE`). When `TRUE`, the framework stamps `node$shortcut_id <- paste0(node$id, "_shortcut")` on every translated source node AND renders the sibling shortcut `textInput` (at `node$shortcut_id`) for you — an "Optional dataset name" box beside the source widget (ADR 0025 item #7). Your `build_ui` renders ONLY the data-payload widget at `node$id` (or `NULL` for a shortcut-only source); it must NOT render the shortcut input itself. Both inputs participate in the runtime substitution cycle: one at `node$id` (the data payload) and one at `node$shortcut_id` (a typed-in name that resolves a `data.frame` from the caller-supplied `envir`). The shortcut value reaches `resolve_data` / `resolve_expr` through `node`. Most sources do not need it — one bound input is the common case. The built-in `ppUpload` sets `shortcut = TRUE`: the file contents bind to `node$id`, the user-typed name binds to `node$shortcut_id`, and the substitution uses the name as the symbol inserted into the generated code. The framework also re-renders the source widget when the shortcut goes non-empty, clearing a stale display (ADR 0025 item #7). The reserved shared key `"shortcut"` is rejected at translate time (see ADR 0025 §1).

`parse_positional_arg`, `parse_named_args` See `ptr_define_placeholder_value()`. Source placeholders use the same arg-schema slots.

`embellish_eval` Optional `function(...)` body used when the placeholder is called as a plain-R function (outside `ptr_app()`). `NULL` (default) supplies a guard that aborts with

a message naming the keyword and noting the call is only meaningful inside `ptr_app()` — source placeholders typically have no out-of-app meaning (a file upload widget cannot produce data at the REPL). Override only if the source has a sensible plain-R interpretation.

Value

The runtime callable. Default for a source placeholder is a guard that aborts when called outside an app context. Also called for its registration side effect; use `ptr_clear_placeholder()` to remove the entry.

`spec=` round-trip

The `spec=` mechanism (see `ptr_app()`) captures a sparse snapshot of input values so the preserve-mode panel can publish a reproducible boot state. For a source placeholder, ONE of two patterns must hold:

- **Shortcut pattern** — set `shortcut = TRUE`. The shortcut input's text value (typically the typed dataset name) carries the round-trip identity; the source's own value at `node$id` is dropped from the spec, because it is typically a per-session Shiny artifact (a `fileInput()` data.frame whose `datapath` is a tempfile path that does not survive the session). The built-in `ppUpload` uses this.

  Data-loading entry point (ADR 0024). When `shortcut = TRUE`, the shortcut sibling is more than a name override for an uploaded frame — it is a typed-in shortcut for loading a `data.frame` from the embedder's environment (`envir` passed to `ptr_app()` / `ptr_server()`). Any valid R name typed into the shortcut input (or seeded via `spec = list(<shortcut-id> = "df_name")`) is looked up via `get(name, envir, inherits = TRUE)` and bound as the resolved source frame, with OR without `default=` on the placeholder. The downstream pipeline, generated code panel, and consumer pickers all read the named frame from `state$eval_env` as if it had been uploaded. Failures surface on the inline error pane via `set_resolve_error`: `"Object 'x' not found in environment."` / `"Object 'x' is not a data frame."` Lookup uses `inherits = TRUE`, so package exports become reachable — typing `"plot"` will resolve to `graphics::plot` and then fail the "is not a data frame" check (loudly, not silently).
- **Scalar pattern** — `shortcut = FALSE`. The widget's value at `node$id` must be a literal that round-trips through `deparse()` — a length-1 string / number / logical, or a simple atomic vector. The `selectInput`-style example above qualifies (its value is a single string).

Source widgets whose primary value is a complex object (raw `fileInput()` data.frame, environment, S4 instance, etc.) without `shortcut = TRUE` cannot round-trip; opt into `shortcut = TRUE` and the framework renders the sibling `textInput(node$shortcut_id, ...)` that carries the binding name, mirroring `ppUpload`.

See Also

`ptr_define_placeholder_value()`, `ptr_define_placeholder_consumer()`, `ptr_clear_placeholder()`.

Examples

```
# A minimal in-memory dataset source (picks from pre-loaded data frames).
ptr_define_placeholder_source(
```

```
  keyword = "dataset",
  build_ui = function(node, label, ...) {
    shiny::selectInput(node$id, label = label,
                       choices = c("mtcars", "iris"))
  },
  resolve_data = function(value, node, ...) {
    if (length(value) != 1L || !nzchar(value)) return(NULL)
    get(value, envir = as.environment("package:datasets"))
  }
)
ptr_clear_placeholder("dataset")
```

`ptr_define_placeholder_value` *Define a value placeholder*

Description

Register a new keyword (e.g. `pct`, `color`, `date`) that ggpaintr will recognise as a substitutable token in a formula. The keyword's widget is built by `build_ui`; the widget's value is turned back into the R code spliced into the rendered call by `resolve_expr`. See `vignette("ggpaintr-tutorial")` § "Defining your own placeholders" for the lifecycle walk-through, signatures table, and runnable `ptr_app()` examples — this help page is reference.

Usage

```
ptr_define_placeholder_value(
  keyword,
  build_ui,
  resolve_expr,
  validate_session_input = NULL,
  parse_positional_arg = NULL,
  parse_named_args = list(),
  embellish_eval = NULL,
  ui_text_defaults = list(label = "Enter a value for {param}")
)
```

Arguments

`keyword`	Single non-empty string. Must be a syntactically valid R name (passes `make.names()`) and not an R reserved word. This is the token users type in the formula, e.g. `geom_point(alpha = pct)`.
`build_ui`	`function(node, label = NULL, selected = NULL, named_args = list(), ...)` returning a Shiny tag. Pass `node$id` as the underlying widget's `inputId`. Read `node$keyword` and `node$param` if you need them. The framework also passes any `ui_text_defaults` field you declare by name (`help`, `placeholder`, `empty_text`) — or accept a `copy = NULL` list and read them off it. If you declare a `named_args` formal (or accept `...`), the framework injects the validated `parse_named_args`

values as a named list — so `parse_named_args = list(step = ptr_arg_numeric())` makes `ppPct(step = 5)` arrive as `named_args$step` (the canonical post-validator value, not the raw call). Every injected argument is opt-in: the injector checks your formals, so name the ones you use and keep `...` to swallow the rest. Always end the signature with `....`

Widget-seeding contract — `selected` (authoritative; the four `invoke_build_ui` call sites in `R/paintr-server.R` implement what is described here, with short pointers back to this docblock).

Declare `selected = NULL` as an explicit formal, **not** a no-default `selected` (or accept `...`). The framework calls `build_ui` on every renderUI fire and delivers `selected` per this precedence:

- First render, `spec=` seed present → seed value.
- First render, no spec, positional default present (e.g. `ppPct(50)`) → `node$default`.
- First render, no spec, no positional default → framework **omits** `selected`; your `selected = NULL` formal default applies.
- Subsequent renders (Update Plot click, upstream change, layer toggle, ...) → `current-input` verbatim. The `spec=` seed is **boot-only**: it wins on the first render only and never participates again, so an upstream-triggered re-render can never snap the widget back to the seed and away from the user's live pick. If the user emptied the widget, `current-input` is whatever the widget emits on clear — one of `NULL`, `character(0)`, `""`, `NA_real_`, `NA_character_`, depending on the widget — and the framework coerces `NULL` to `character(0)` so you always receive a value on subsequent renders.

Because the framework omits the argument on the first-render-no- default path, a hook signature without a formal default for `selected` aborts there with `argument "selected" is missir`

Render empty when `selected` **is empty.** Treat any of `NULL`, `character(0)`, `""`, `NA_real_`, `NA_character_` as "the user wants this widget empty" and render the widget empty. Do not substitute a hardcoded fallback inside `build_ui` for the empty case — that re-introduces the deselect-snaps-back-to-default defect the framework's seeding layer is designed to prevent (closure-flag in `ptr_setup_value_uis()` / `ptr_setup_consumer_uis()` / `ptr_bind_shared_consumer_uis()`). Boot-time defaults belong in the formula, e.g. `ppPct(50)`, and reach you through `selected`, not through a hardcoded constant in the hook body.

Never read `node$default` **directly.** It is exposed only so the framework can route it into `selected` on the boot fire. Reading it inside the hook bypasses the precedence chain and the closure- flag persistence guards, and will re-clobber a user-cleared widget on every Update Plot click.

`resolve_expr` `function(value, node, ...)` returning the R expression spliced into the rendered call. `value` is `input[[node$id]]` — whatever Shiny stores for that widget. Allowed return types: scalar atomic (numeric / character / logical / integer), `name/symbol` (build with `rlang::sym()`), `call/language` (build with `rlang::ppExpr()`), or `NULL` to **prune the argument** from the rendered call. Use `NULL` for empty / not-yet input; throw with `rlang::abort()` for malformed input.

`validate_session_input`

Optional `function(value, ctx)` called before `resolve_expr`. Return `TRUE`

/ NULL to accept; return a single character string to reject (surfaced inline as the error message, layer pruned). ctx is a plain list with named fields: node (the placeholder AST node, carries $id, $keyword, $args), keyword (convenience alias for node$keyword), upstream_cols, and data. For a *value* placeholder, ctx$upstream_cols and ctx$data are **always** NULL — value placeholders have no upstream column scope by definition. They are present in the signature so the same validator shape works across all roles; see ptr_define_placeholder_consumer() for the data-aware role where those fields are populated. ggpaintr invokes this function as validate_input(value, ctx) — no other positional or named arguments are passed, and ctx carries exactly the four fields above. The signature does not require ....

parse_positional_arg
: Optional validator closure for the (single) positional argument the keyword accepts inside a formula. NULL (default) means positional arguments are rejected at translate time. A function receives the unevaluated AST and must return a canonical value or rlang::abort(). Validators are expected to operate on the AST only and not call eval(); ggpaintr trusts the author. Authors who eval in a validator are opting into the risk of running user code at translate time.

parse_named_args
: Named list of validator closures for additional named arguments beyond the reserved shared = ... Each entry's closure receives the unevaluated AST and returns a canonical value or rlang::abort(). The canonical values are delivered to build_ui as its named_args list (and are also carried on node$named_args, so resolve_expr / validate_session_input can read them off node). Default is list() (no named args). The name "shared" is reserved and may not appear here.

embellish_eval
: Optional function(x, ...) body used when the placeholder keyword is *also* called as a plain-R function (outside a formula context) — i.e. how a placeholder-embellished ggplot expression evaluates with no app. When NULL (default), the identity from embellish_identity() (function(x, ...) x) is supplied — calling pct(0.5) returns 0.5 unchanged. Override to give the keyword a non-identity plain-R meaning (e.g. embellish_symbol_to_string()). The same callable is returned to the caller of this helper so authors can bind it under the same name as the keyword: ppPct <- ptr_define_placeholder_value(keyword = "ppPct", ...).

ui_text_defaults
: Named list of single non-NA character defaults feeding the ui_text tree. Allowed names: label, help, placeholder, empty_text. Strings may contain {param}, which is interpolated to the surrounding formal-argument name at render time.

Details

Three roles. Pick this constructor for a *value* placeholder (a self-contained widget like a slider, color picker, or numeric input). Use ptr_define_placeholder_consumer() when the widget needs the upstream column names (column pickers). Use ptr_define_placeholder_source() when the widget *produces* the data the rest of the formula reads from (file upload, dataset chooser).

Value

The runtime callable. Default for a value placeholder is the identity `function(x, ...) x`; override with `embellish_eval = ...`. The helper is also called for its registration side effect — use `ptr_clear_placeholder()` to remove the entry.

See Also

`vignette("ggpaintr-tutorial")` § "Defining your own placeholders"; `ptr_define_placeholder_consumer()`, `ptr_define_placeholder_source()`, `ptr_clear_placeholder()`.

Examples

```
# A percentage placeholder: user types a number 0-100; we splice
# the fraction 0-1 into the rendered call. The hook reads `selected`
# to honor a formula default like `pct(75)` at boot and to keep a
# user-cleared widget empty across Update Plot fires (do NOT
# substitute a hardcoded fallback when selected is empty).
ptr_define_placeholder_value(
  keyword = "pct",
  build_ui = function(node, label = NULL, selected = NULL, ...) {
    n <- suppressWarnings(as.numeric(selected))
    initial <- if (length(n) == 1L && is.finite(n)) n else NA_real_
    shiny::numericInput(node$id, label = label %||% "Percent",
                        value = initial, min = 0, max = 100, step = 1)
  },
  resolve_expr = function(value, node, ...) {
    if (length(value) != 1L || !is.finite(value)) return(NULL)
    value / 100
  },
  parse_positional_arg = ptr_arg_numeric(),  # accept ppPct(50)-style defaults
  ui_text_defaults = list(label = "Percent for {param}")
)
ptr_clear_placeholder("pct")
```

`ptr_extract` *Extract Runtime Outputs From a* `ptr_state`

Description

Read the latest plot object, error message, or generated code text from the runtime result stored on a `ptr_state`. Use these to compose custom UIs or to test the runtime in `shiny::testServer`.

Usage

```
ptr_extract_plot(state)

ptr_extract_error(state)

ptr_extract_code(state)
```

Arguments

state	A ptr_state from ptr_init_state().

Value

ptr_extract_plot returns a ggplot object (or NULL on failure); ptr_extract_error returns a string or NULL; ptr_extract_code returns a single string.

Reactive contexts

Each function wraps its read in shiny::isolate(), so it works in both reactive and non-reactive contexts and returns the current value without establishing a reactive dependency.

Do not call these inside a render*{} **block** if you want the output to update when the plot rerenders. Because isolate() suppresses the dependency on state$runtime(), the render block fires once on mount and never again. Inside a reactive context, read state$runtime() directly — that takes the dependency and re-fires on every *Update plot* click. Reserve ptr_extract_* for non-reactive contexts: download handlers, shiny::testServer() assertions, and one-shot reads outside any session.

Examples

```
shiny::isolate({
  state <- ptr_init_state(
    "ggplot(mtcars, aes(x = wt, y = mpg)) + geom_point()"
  )
  ptr_extract_code(state)
})
```

---

ptr_ggplotly *Build a plotly Widget From a* ggpaintr *Instance, Wired for Linked Selection*

Description

ptr_ggplotly() is the state-first counterpart to plotly::ggplotly() for the supported L3 custom-render pattern (see ptr_server(), section "Custom render (L3)"). Call it from inside plotly::renderPlotly() on the live instance returned by ptr_server(): it reads the drawn plot off state$runtime()$plot, mints a per-draw row key (the plotly key aesthetic, on a widget-only copy of the data), sets dragmode = "select", registers the plotly_selected / plotly_deselect events, and returns a plain plotly object so your own plotly verbs stay composable after it.

Usage

```
ptr_ggplotly(state, ..., source = NULL)
```

Arguments

state	A ptr_state handle as returned by ptr_server().
...	Forwarded verbatim to plotly::ggplotly() (e.g. tooltip = "all").
source	Character scalar for plotly's source = channel, or NULL (default) to derive a distinct string from the instance namespace.

Details

The helper owns the shiny::req() pre-draw guard (it raises Shiny's silent pre-draw condition before the first draw, exactly like a bare req(p)) and derives plotly's source = channel from the instance namespace, so the companion selection reader coordinates without extra wiring. The user's drawn data is never mutated: keys are minted per draw on the widget copy only.

plotly is an optional dependency (in Suggests). When it is not installed, this function aborts, mirroring ptr_app_bslib()'s bslib guard.

Value

A plain plotly htmlwidget object (inherits class "plotly"), with dragmode = "select" set and the plotly_selected / plotly_deselect events registered.

See Also

ptr_server() for the L3 custom-render contract; plotly::ggplotly().

Examples

```
if (interactive()) {
library(shiny)
library(plotly)
f <- rlang::expr(
  ggplot(mtcars, aes(x = ppVar(wt), y = ppVar(mpg))) + geom_point()
)
ui <- ptr_ui_page(
  ptr_ui_controls(formula = f, id = "p"),
  plotly::plotlyOutput("plt")
)
server <- function(input, output, session) {
  state <- ptr_server(f, "p")
  output$plt <- plotly::renderPlotly({
    # state-first: req() guard + key minting + select wiring are internal
    ptr_ggplotly(state, tooltip = "all") |>
      plotly::layout(legend = list(orientation = "h"))
  })
}
shinyApp(ui, server)
}
```

ptr_gg_extra *Add* ggplot2 *Layers Programmatically*

Description

Evaluate one or more ggplot2 expressions and attach the results as "extras" on the state. Extras are folded into the plot during the next runtime cycle when `state$runtime()$ok` is `TRUE`. Eval failures leave the existing extras untouched.

Usage

```
ptr_gg_extra(state, ...)
```

Arguments

`state`	A `ptr_state` from `ptr_init_state()`.
`...`	ggplot2 layer expressions (e.g. `ptr_gg_extra(state, ggplot2::scale_x_log10(), theme_minimal())`). Captured unevaluated and stored as quosures, then evaluated in `state$eval_env`. Eval errors propagate and leave the existing extras untouched (atomic update).

Value

`state`, invisibly.

Examples

```
shiny::isolate({
  state <- ptr_init_state(
    "ggplot(mtcars, aes(x = wt, y = mpg)) + geom_point()"
  )
  state <- ptr_gg_extra(state, ggplot2::theme_minimal())
  ptr_extract_code(state)
})
```

ptr_id_table *Enumerate every Shiny id a ggpaintr formula produces*

Description

Walks the typed AST of a single formula and returns one row per Shiny id ever rendered by `ptr_ui()` / `ptr_server()` for that formula: placeholder sleeves, inner widgets, source companions, layer-level controls (checkbox, subtab, content container), pipeline stage toggles, plus the static infrastructure ids (`ptr_plot`, `ptr_error`, `ptr_code`, `ptr_code_mode`, `ptr_update_plot`).

Usage

```
ptr_id_table(formula, id = NULL)
```

Arguments

formula	A single ggpaintr formula string (the same input you would pass to ptr_app() / ptr_server()).
id	Optional outer namespace — the same string you pass to ptr_server(formula, id = ...). When supplied, rows whose scope is "instance" come back with that namespace prefixed (<id>-<raw>); "global" rows stay bare. When NULL, every row shows its bare id.

Details

Advanced (L3) users call this to build their own UI by hand and still get the server's bindings to match. The default-layout L2 path does not need it; ptr_ui() emits these ids internally.

Value

A data.frame with one row per Shiny id and ten columns:

- id — the Shiny id (prefixed when id= is given and scope == "instance").
- kind — "input_widget" or "output_slot".
- role — semantic role: "placeholder", "source_companion", "layer_checkbox", "layer_subtab", "layer_content", "stage_enabled", "ptr_plot", "ptr_error", "ptr_code", "ptr_code_mode", "ptr_update_plot".
- scope — "instance" (namespaced via shiny::NS(id)) or "global" (un-namespaced; only used by cross-formula shared-panel keys in a ptr_shared() setup — see Single-formula section).
- include_in_ui — TRUE when the row is something the user *places* in their custom UI; FALSE when the server populates it inside a sleeve (<id>_ui) and the user must not place it manually. The two FALSE cases are the inner placeholder widget and ppUpload's file-name companion.
- layer — layer name ("ggplot", "geom_point", ...) or NA.
- keyword — placeholder keyword ("ppVar"/"ppText"/...) or NA.
- param — argument or aesthetic name ("x", "color", "data", ...) or NA.
- parent_call — immediate enclosing call ("aes", "head", or the layer itself when the placeholder is a direct layer arg) or NA.
- shared — shared key ("xcol", ...) or NA.

Stability under formula edits

Adding a layer at the end keeps existing ids stable. Reordering arguments inside aes() or a pipeline shifts positional paths and therefore changes ids; renaming a placeholder keyword or its shared= annotation also changes ids.

Single-formula

ptr_id_table() accepts a single formula. In multi-instance (ptr_shared(formulas = list(. . .))) layouts the partition rule decides whether shared_<key> lives in an instance's inline section or the cross-formula panel. The scope column reflects the single-formula interpretation ("instance"); when embedding into a shared-panel context those rows are bare ("global") and you should override accordingly.

Examples

```
ptr_id_table("ggplot(ppUpload, aes(x = ppVar, y = ppVar)) + geom_point(color = ppText)")
ptr_id_table("ggplot(ppUpload, aes(x = ppVar))", id = "myplot")
```

---

ptr_init_state *Construct the ggpaintr runtime state container*

Description

Builds the ptr_state object — the translated typed AST (as a reactiveVal), the runtime result, the per-layer resolved-data caches, the eval environment, the input-snapshot machinery, and the shared bindings / draw trigger — used by ptr_server() and the advanced-embedder accessors (ptr_extract_plot() / ptr_extract_error() / ptr_extract_code(), ptr_gg_extra()).

Usage

```
ptr_init_state(
  formula,
  envir = parent.frame(),
  ui_text = NULL,
  expr_check = TRUE,
  safe_to_remove = character(),
  shared = list(),
  draw_trigger = NULL,
  producer_debounce_ms = NULL,
  ns = shiny::NS(NULL),
  server_ns = ns,
  auto_bind_shared = FALSE,
  shared_resolutions = list(),
  shared_stage_enabled = list(),
  panel_sources = list(),
  plots = NULL
)
```

Arguments

formula	A single formula string with ggpaintr placeholders.
envir	Environment used to resolve local data objects.

`ui_text` Optional named list of copy overrides; see `ptr_ui_text()` for the full schema and current defaults.

`expr_check` Controls formula-level `ppExpr` validation: `TRUE` (default) applies the built-in denylist + AST walker; `FALSE` disables formula-level validation; a `list` with `deny_list`/`allow_list` entries customises the formula-level policy. Runtime-typed `ppExpr` input is always screened against the built-in denylist regardless. See the safety chapter of the ggpaintr book (development-version docs): https://willju-wangqian.github.io/ggpaintr-book/safety.html.

`safe_to_remove` Character vector of additional function names whose zero-argument calls should be dropped after substitution.

`shared` Named list of reactives (one per shared key) supplied by an outer wrapper such as `ptr_app_grid()`. Defaults to `list()`.

`draw_trigger` Optional reactive carrying a click counter — a numeric scalar that is `>= 1` once its button has been clicked (e.g. the grid app's "Draw all" `input$...` value). A redraw fires only when the carried value looks clicked; a reactive carrying any other value (a data frame, a timestamp) invalidates the runtime observer but never triggers a draw. To redraw on arbitrary reactive changes (e.g. a reactive pipeline head in the formula), use `ptr_options(gate_draw = FALSE)` live mode instead. Defaults to `NULL`.

`producer_debounce_ms`
Optional. Controls the debounce window applied to producer-style placeholder inputs (`ppText`, `ppNum`, `ppExpr`) before they invalidate downstream consumer caches. `NULL` (default) enables auto mode: window starts at 0 ms and the runtime flips to 300 ms after three consecutive upstream resolutions exceed 150 ms (and back to 0 after five consecutive resolutions under 80 ms). Pass `0` to force off forever, or a positive integer to pin a manual window.

`ns` A namespace function used for rendered ids (UI side).

`server_ns` A namespace function used for server-side input lookups. Defaults to `ns`.

`auto_bind_shared`
If `TRUE`, the host (single-plot `ptr_app()` or `ptr_app_grid()` auto-render path) binds shared widgets at host scope. Relaxes the "missing-from-bindings" check in `ptr_validate_shared_bindings()` (the host auto-binds instead).

`shared_resolutions`
Named list (keyed by raw shared key) of host-computed resolutions for shared data-consumer (`ppVar`) widgets, as returned by `ptr_resolve_shared_consumers()`. When an entry is present, the runtime validates that key's selection against the host-resolved upstream (the same data the host picker was built from) instead of the per-layer `node$upstream`, so a value valid in the host picker is never rejected by one layer's narrower upstream. Defaults to `list()` (no host resolutions; per-layer behaviour).

`shared_stage_enabled`
Named list (keyed by raw shared key) of reactives, each returning a logical, that toggle the orphan pipeline stages owned by that shared key (as carried in a `ptr_shared_server()` bundle). A missing or unset entry leaves the stage enabled. Defaults to `list()`.

panel_sources — Named list (keyed by source-node id) of reactives, each returning the host-loaded data for a panel-owned shared source (ADR 0023). Populated by the host's `ptr_setup_panel_sources()` and threaded through a `ptr_shared_server()` bundle so per-instance binders read the host's primed data (`state$panel_sources[[node$id]]`) instead of re-wiring their own source UI. Defaults to `list()` (single-plot / per-instance context — no panel-owned sources).

plots — Optional list of formula strings for grid contexts. When supplied (typically by `ptr_app_grid()`), the validator for `shared` bindings cross-checks shared-key references against every plot's placeholder set, so a `shared = "..."` annotation that exists in plot 2 but not plot 1 still validates. Defaults to `NULL` (single- plot context — only `formula` is checked).

Details

This is a *state container*, not a from-scratch reactive-app builder: it allocates the reactives but does not attach the pipeline / runtime observers (those live in internal `ptr_setup_*` helpers wired by `ptr_server()`). Reach for `ptr_init_state()` directly when you want to drive the typed tree programmatically or exercise ggpaintr under `shiny::testServer()`; for a fully wired app use `ptr_server()`.

Value

A `ptr_state` list (S3 class `c("ptr_state", "list")`).

Examples

```
shiny::isolate({
  state <- ptr_init_state(
    "ggplot(mtcars, aes(x = ppVar, y = ppVar)) + geom_point()"
  )
  is.list(state)
})
```

---

ptr_llm_primer — *Get the ggpaintr LLM system-prompt primer*

Description

Returns the text of `inst/llm/primer.md` as a single string. Intended for use as the `system_prompt =` (or equivalent) argument when wiring ggpaintr into an LLM client such as `ellmer`, so the model knows when to reach for ggpaintr and at which of the three integration levels.

Usage

```
ptr_llm_primer()
```

Details

The primer is short by design — it establishes the extensibility model and points the model at `ptr_llm_topic()` for runnable examples. A companion tool that exposes `ptr_llm_topic()` to the model lets it pull only the example it needs, instead of loading every topic into the system prompt.

Value

A single character string.

See Also

`ptr_llm_topic()`, `ptr_llm_topics()`

Examples

```
primer <- ptr_llm_primer()
cat(substr(primer, 1, 200))
```

---

ptr_llm_register *Register ggpaintr with an ellmer chat session*

Description

One-line registration of the ggpaintr docs tool on an existing ellmer `Chat` object. The tool wraps `ptr_llm_topic()` so the LLM can pull focused, runnable examples on demand instead of loading every topic into the system prompt.

Usage

```
ptr_llm_register(chat, tool_name = "ggpaintr_docs")
```

Arguments

`chat`	An ellmer `Chat` object (from `ellmer::chat_anthropic()`, `ellmer::chat_openai()`, etc.).
`tool_name`	String. The name the LLM will call the tool under. Defaults to `"ggpaintr_docs"`; override if it would collide with another tool you already registered.

Details

The set of valid topic names is snapshotted at registration time using `ptr_llm_topics()`, so the LLM cannot request a topic that does not exist. If you upgrade ggpaintr in the same session and new topics are added, call this again on a fresh chat.

This function does *not* set the chat's system prompt. Pass `ptr_llm_primer()` to the `system_prompt =` argument of your `chat_*()` constructor so the model knows when to reach for the tool.

Value

The chat object (invisibly), for piping.

See Also

ptr_llm_primer(), ptr_llm_topic(), ptr_llm_topics()

Examples

```
if (interactive()) {
library(ellmer)
chat <- chat_anthropic(system_prompt = ptr_llm_primer())
ptr_llm_register(chat)
chat$chat("Build a Shiny app where the user picks X and Y columns from mtcars.")
}
```

---

ptr_llm_topic *Fetch a ggpaintr LLM topic*

Description

Returns the runnable example + commentary for one topic as a single character string. Designed to back an LLM tool such as ggpaintr_docs(topic): the model calls it when the user asks for help with an interactive ggplot task, and receives exactly one focused example instead of the entire manual.

Usage

```
ptr_llm_topic(topic)
```

Arguments

topic	Topic name. Must be one of ptr_llm_topics().

Details

Each topic is derived from (and kept in sync with) either README.Rmd or the tutorial vignette (ggpaintr-tutorial).

Value

A single character string.

See Also

ptr_llm_topics(), ptr_llm_primer()

Examples

```
cat(ptr_llm_topic("level1_ptr_app"))
```

ptr_llm_topics *List available ggpaintr LLM topic names*

Description

Returns the character vector of topic names accepted by `ptr_llm_topic()`. Matches the files in `inst/llm/topics/` (stripped of the `.md` extension), sorted alphabetically.

Usage

```
ptr_llm_topics()
```

Value

A character vector.

See Also

`ptr_llm_topic()`, `ptr_llm_primer()`

Examples

```
ptr_llm_topics()
```

ptr_normalize_column_names *Normalize Dataset Column Names for* ggpaintr

Description

Normalize incoming column names so `ppVar` placeholders can require exact, syntactic, unique column-name matches at runtime.

Usage

```
ptr_normalize_column_names(data)
```

Arguments

`data`	A data frame or an object coercible with `as.data.frame()`.

Details

Uploaded data is normalized automatically by the runtime — every successful ppUpload placeholder passes through this function (or its data.frame-coercing sibling for non-data.frame returns from readRDS / readxl / jsonlite). Call ptr_normalize_column_names() yourself only for **in-session** data frames you reference by name from a formula. If a local data frame has spaces, reserved words, or duplicates in its column names, pipe it through this function before passing it to ptr_app().

Value

A tabular object with ggpaintr-safe column names. Existing data.frame subclasses keep their class. Names are made syntactic, unique, and safe against reserved-word collisions. Non-data.frame inputs return the data.frame created by as.data.frame().

Examples

```
messy <- data.frame(
  check.names = FALSE,
  "first column" = 1:3,
  "if" = 4:6
)

clean <- ptr_normalize_column_names(messy)
names(clean)
```

---

ptr_options *Get or Set ggpaintr Package Options*

Description

Combined getter/setter for ggpaintr's global settings, modeled after base base::options(). Calling with no arguments returns all current settings as a named list. Passing one or more named logical arguments sets those settings and invisibly returns the previous values, suitable for the do.call(ptr_options, old) round-trip pattern used by withr::with_options() and on.exit().

Usage

```
ptr_options(...)
```

Arguments

...	Named logical arguments — one per setting to update. Setting names must match the registry above.

Value

When called with no arguments, a named list of all current setting values. When called with named arguments, the previous values of the updated settings, returned invisibly.

Available settings

`verbose` Logical. When `TRUE`, ggpaintr emits informative messages such as "Layer foo() removed (no arguments provided)." Default `FALSE` — these messages are intended for debugging the formula pipeline and are off by default. Underlying option: `options(ggpaintr.verbose = ...)`.

`gate_draw` Logical. When `TRUE` (the default), the rendered plot updates only when the user clicks the "Update plot" button — every placeholder change is batched until the click. When `FALSE`, the button is omitted from the UI and the plot re-renders reactively on every placeholder change (live mode). Read once when the app is built, so set it before calling `ptr_app()` / `ptr_ui()`. Underlying option: `options(ggpaintr.gate_draw = ...)`.

`suppress_warnings` Logical. When `TRUE`, R warnings emitted while the plot is drawn (e.g. `loess` fit warnings such as "all data on boundary of neighborhood" or "Failed to fit group N") are silenced rather than printed to the console. Default `FALSE` — warnings surface as usual. Only the plot-drawing step is wrapped; errors still propagate to the inline error pane. Read once when the app is built, so set it before calling `ptr_app()` / `ptr_ui()`. Underlying option: `options(ggpaintr.suppress_warnings = ...)`.

Examples

```
# Inspect current values
ptr_options()

# Silence the "Layer ... removed" notice for one block
old <- ptr_options(verbose = FALSE)
on.exit(do.call(ptr_options, old), add = TRUE)
```

`ptr_plotly_selection` *Read a* `ptr_ggplotly()` *Linked Selection As Rows or a Flag Column*

Description

`ptr_plotly_selection()` is the companion reader for `ptr_ggplotly()`: it returns a Shiny reactive carrying the current brush/lasso selection of the plotly widget, projected one of two ways. Call it once in your server (not inside a render) on the live instance returned by `ptr_server()` and feed the returned reactive to a table, a second formula, or any downstream consumer.

Usage

```
ptr_plotly_selection(state, mode = c("rows", "flag"), source = NULL)
```

Arguments

`state`	A `ptr_state` handle as returned by `ptr_server()`.
`mode`	`"rows"` (default) or `"flag"` — which projection of the one selection to return. Any other value aborts (the bare-indices projection is rejected by design).
`source`	Character scalar for plotly's `source =` channel, or `NULL` (default) to derive the same distinct string `ptr_ggplotly()` derives from the instance namespace.

Details

A selection names rows of the **drawn data** — `state$runtime()$plot$data` of the draw that produced the widget — never "the original object". Keys are minted per draw and are meaningless across draws, so the selection **resets to empty on every draw** (a redraw after a `filter()`/`mutate()` pipeline-head change or an upload swap starts the selection over). A `plotly_deselect` event clears it likewise. Two projections of the one selection:

- `mode = "rows"` — the selected slice; a **zero-row data frame with the drawn data's columns** when nothing is selected (so a `shiny::renderTable()` consumer needs no `req()` dance).
- `mode = "flag"` — the **full** drawn data plus a logical `.ptr_selected` column, `TRUE` exactly at the selected rows (all `FALSE` when empty).

`.ptr_selected` is a reserved name: a pre-existing `.ptr_selected` on the drawn data is silently overwritten (the overwrite keeps chained selection-fed instances correct). The internal `.ptr_row` key never appears in either projection.

Pre-draw window: before the first draw there is no snapshot yet, so the returned reactive raises Shiny's silent pre-draw condition (via `shiny::req()`); under live mode a selection-fed instance shows its inline pre-draw state for one flush. **Live-mode key reset:** changing a placeholder widget re-draws the source plot, which re-mints the keys, so the selection resets to empty on that picker change.

plotly is an optional dependency (in `Suggests`); this reader is only meaningful alongside `ptr_ggplotly()`, which guards on plotly being installed.

Value

A Shiny reactive. Its value is the rows projection (a data frame, the selected slice; zero rows, same columns, when empty) or the flag projection (the full drawn data plus a logical `.ptr_selected`), per `mode`.

See Also

`ptr_ggplotly()` for the widget side; `ptr_server()` for the L3 custom-render contract.

Examples

```
if (interactive()) {
library(shiny)
library(plotly)
f <- rlang::expr(
  ggplot(mtcars, aes(x = ppVar(wt), y = ppVar(mpg))) + geom_point()
)
ui <- ptr_ui_page(
  ptr_ui_controls(formula = f, id = "p"),
  plotly::plotlyOutput("plt"),
  tableOutput("sel")
)
server <- function(input, output, session) {
  state <- ptr_server(f, "p")
  output$plt <- plotly::renderPlotly({
    ptr_ggplotly(state, tooltip = "all")
```

```
  })
  # Full loop: brush the plot -> the selected rows appear in the table.
  sel <- ptr_plotly_selection(state, mode = "rows")
  output$sel <- renderTable(sel())
}
shinyApp(ui, server)
}
```

`ptr_resolve_ui_text` *Resolve copy for one ggpaintr control or app element*

Description

Looks up the effective label/help/placeholder for a single UI element, applying the `defaults` -> `params` -> `layers` specificity chain and interpolating the `{param}` / `{layer}` tokens. Placeholder authors can call this inside a custom `build_ui` hook so their control is labelled through the same override chain as the built-in controls.

Usage

```
ptr_resolve_ui_text(
  component,
  keyword = NULL,
  param = NULL,
  layer_name = NULL,
  ui_text = NULL
)
```

Arguments

`component`	One of `title`, `draw_button`, `draw_all_button`, `layer_picker`, `data_subtab`, `controls_subtab`, `upload_file`, `upload_name`, `layer_checkbox`, or `control` (for a placeholder control, in which case `keyword` is required).
`keyword`	Placeholder keyword (e.g. `"ppVar"`, `"ppNum"`); required when `component = "control"`.
`param`	Optional parameter / aesthetic name (e.g. `"x"`); only used when `component = "control"`.
`layer_name`	Optional layer name (e.g. `"facet_wrap"`); only used when `component = "control"`, to pick up a `layers$<layer>$...` override.
`ui_text`	`NULL`, a list of overrides, or an already-merged `ptr_ui_text` object (see `ptr_ui_text()`).

Value

A named list with `label`, `help`, `placeholder`, and `empty_text`.

Examples

```
# Resolve copy for the title element
ptr_resolve_ui_text("title")

# Resolve copy for a var control on the x-axis
ptr_resolve_ui_text("control", keyword = "ppVar", param = "x")

# Inside a custom `build_ui` hook, label the control through the same
# override chain ggpaintr uses for built-in controls:
my_build_ui <- function(node, label, ...) {
  copy <- ptr_resolve_ui_text(
    "control",
    keyword = node$keyword,
    param = node$param,
    ui_text = list(...)$ui_text
  )
  shiny::textInput(node$id, label = copy$label %||% label)
}
```

`ptr_server` *Server for a* `ggpaintr` *Formula*

Description

The **single public server-side entry point** for a `ggpaintr` formula, used identically at L2 (default layout via `ptr_ui()`) and L3 (your own hand-composed layout from the bare `ptr_ui_*` pieces). It namespaces and wires the whole reactive engine, registers the built-in `ptr_plot` / `ptr_error` / `ptr_code` outputs (a piece you never place in the UI is a harmless no-op), and **returns the** `ptr_state` so you can drive a custom renderer. Additional arguments are forwarded to `ptr_init_state()` (e.g. `shared`, `draw_trigger`, `expr_check`, `safe_to_remove`, `ui_text`).

Usage

```
ptr_server(
  formula,
  id = NULL,
  envir = parent.frame(),
  ...,
  shared_state = NULL,
  spec = NULL
)
```

Arguments

`formula`	Either a single character scalar containing a ggplot expression with `ggpaintr` placeholders, or an unquoted ggplot expression supplied directly. See `ptr_app()` for the full contract (expression capture via `rlang::enexpr()`, symbol resolution, wrapper unwrap, and the native-pipe caveat in expression mode).

id	Optional module id; must match the id passed to ptr_ui() or to the bare L3 pieces. Defaults to NULL (identity namespace, single-instance use).
envir	Environment used to resolve local data objects.
...	Forwarded to ptr_init_state().
shared_state	Optional ptr_shared_state returned by ptr_shared_server(). When supplied, populates shared, draw_trigger, and shared_resolutions defaults. Required when the formula declares a shared = "..." placeholder driven by a cross-formula ptr_shared_panel() and the equivalent ... arguments are not supplied directly.
spec	An optional named list of fully-qualified Shiny input id -> value, used to override widget defaults at session boot.

Value

The ptr_state list from ptr_init_state(). This is the **supported L3 custom-render handle**: state$runtime()$plot / $code / $error. (Its server_ns_fn / ui_ns_fn slots are internal plumbing — not a public escape hatch.)

Custom render (L3)

Custom rendering is **UI-side**: place your own output widget (e.g. plotly::plotlyOutput()) at shiny::NS(id)("my_plot"), never place ptr_ui_plot(), and read the live plot off the returned state — there is **no** user-authored moduleServer wrapping any ggpaintr engine and **no** lower-level server function to reach for:

```
# server:
state <- ptr_server(formula, "p")
output$my_plot <- plotly::renderPlotly(state$runtime()$plot)
# ui: plotly::plotlyOutput(shiny::NS("p")("my_plot"))
```

state$runtime() is reactive; $plot is the built ggplot/ggplot-like object, $code the generated source string, $error any inline error. See vignette("ggpaintr-tutorial").

For cross-formula coordination — multiple ggpaintr instances driven by one widget — build the coordinator with ptr_shared_server() and pass the returned ptr_shared_state as shared_state =. The state's shared / draw_trigger / shared_resolutions slots are unpacked and forwarded to ptr_init_state(); if an explicit shared = ... / draw_trigger = ... / shared_resolutions = ... is also passed via ..., that explicit value wins. A single formula with shared = "..." placeholders needs no shared_state — ptr_server() self-binds every declared key under its own namespace, matching what ptr_app() does.

See Also

ptr_ui(), ptr_ui_plot(), ptr_shared(), ptr_shared_panel(), ptr_shared_server().

Examples

```
if (interactive()) {
  f <- rlang::expr(ggplot(mtcars, aes(x = ppVar, y = ppVar)) + geom_point())
  # L2: default layout
  shiny::shinyApp(
    ui = shiny::fluidPage(ptr_ui(!!f, "p")),
    server = function(input, output, session) {
      ptr_server(!!f, "p")
    }
  )
  # L3: own the render path off the returned state
  shiny::shinyApp(
    ui = ptr_ui_page(
      ptr_ui_controls(formula = !!f, id = "p"),
      plotly::plotlyOutput(shiny::NS("p")("my_plot"))
    ),
    server = function(input, output, session) {
      state <- ptr_server(!!f, "p")
      output[[shiny::NS("p")("my_plot")]] <-
        plotly::renderPlotly(state$runtime()$plot)
    }
  )
}
```

`ptr_shared` *Build the Shared Coordinator for a Multi-Instance Embedding*

Description

Constructs a single pure, non-reactive coordinator object from the full set of plot formulas. The coordinator computes the cross-formula **partition** – a shared key used by exactly one formula renders in that formula's inline shared section; a key used by two or more formulas renders in the one standalone `ptr_shared_panel()`. Because every consumer derives its view from this one object, the UI and server can never disagree about the partition.

Usage

```
ptr_shared(
  formulas,
  id = NULL,
  ui_text = NULL,
  expr_check = TRUE,
  draw_all_label = "Draw all"
)
```

Arguments

formulas	A non-empty character vector or list, one element per embedded ptr_ui() instance. Each element is either a formula **string** or a **quoted ggplot expression** (rlang::expr(...) / quote(...)); quoted expressions are deparsed to their source string, and the two forms are interchangeable (mixed lists are allowed). A built ggplot object (e.g. g <- ggplot(...) + geom_point() then passing g) is **not** accepted – its source is unrecoverable, so quote it with expr() instead. Note: a native pipe |> inside a quoted expression is desugared by R's parser before capture, so it does not survive into the generated code panel (use %>% or a formula string to preserve it).
id	Optional character scalar that namespaces every id this coordinator emits, so two or more coordinators can coexist in one Shiny session without colliding on shared input slots. When non-NULL, every id (shared_<key>, shared_<key>_stage_enabled, ptr_shared_draw_all, ptr_shared_errors, the stage-block DOM id, and each consumer renderUI container) is prefixed <id>- using Shiny NS()'s separator. The default NULL preserves today's flat ids byte-for-byte – single-panel apps need no change. Must be NULL or a non-empty string matching ^[A-Za-z][A-Za-z0-9_]*$; ptr_shared_panel(), ptr_ui_shared_panel(), and ptr_shared_server() all read it off obj$id – their signatures do not change.
ui_text	Optional copy overrides forwarded to the auto-built widgets (see ptr_app()'s ui_text argument).
expr_check	Whether to validate ppExpr placeholders during formula translation. Default TRUE. Runtime-typed ppExpr input is always screened against the built-in denylist regardless.
draw_all_label	Label for the "Draw all" button rendered in the panel when two or more formulas are supplied.

Details

This is strictly multi-instance API: with a single ggpaintr instance all shared widgets auto-render inline and no coordinator is needed. The coordinator is consumed by exactly three functions, each taking only the object: ptr_shared_panel(), ptr_ui_shared_panel(), and ptr_shared_server().

Errors when no formula declares a shared = "..." annotation – building the coordinator is a declaration of intent.

Value

A ptr_shared_spec S3 object. Public fields: panel_keys (cross-formula keys), local_keys_by_formula (per-formula inline keys). Deterministic and idempotent for a given formulas.

Removed shared_ui

shared_ui (a named list of function(id) -> shiny.tag builders) is no longer supported. Its original intent was to let two placeholders that share a key share the same UI settings. But each placeholder already carries its own build_ui rule, and two shared= placeholders simply reuse that one rule when their widget collapses to the panel. The customisation therefore comes from the placeholder's build_ui (ptr_define_placeholder_*(build_ui = ...)), not from a separate

shared_ui override — so the override was redundant, and for var consumers it actively dropped the formula default and froze the column list. To customise a shared widget, define a custom placeholder with the build_ui you want and use it in the formula. Label-only divergence for the shared widget still has its own channel (node$shared_label).

See Also

ptr_shared_panel(), ptr_ui_shared_panel(), ptr_shared_server(), ptr_ui().

Examples

```
obj <- ptr_shared(c(
  "ggplot(mtcars, aes(x = ppVar(shared='x'), y = ppVar)) + geom_point()",
  "ggplot(mtcars, aes(x = ppVar(shared='x'), y = ppVar)) + geom_bar()"
))
obj$panel_keys   # "x" — shared by both formulas

# Quoted expressions work too, and may be mixed with strings:
obj2 <- ptr_shared(list(
  rlang::expr(ggplot(mtcars, aes(x = ppVar(shared = "x"), y = mpg)) + geom_point()),
  "ggplot(mtcars, aes(x = ppVar(shared = 'x'), y = hp)) + geom_line()"
))
obj2$panel_keys  # "x"
```

---

ptr_shared_panel *Render the Standalone Shared Panel (L2, Self-Contained)*

Description

Renders the single page-level shiny::wellPanel() holding exactly the coordinator's cross-formula keys (obj$panel_keys, referenced by two or more formulas). The panel is self-contained: it owns its .ptr-app theming scope and the bundled ggpaintr asset dependency, so it can be dropped straight into a host layout. Its server counterpart ptr_shared_server() must run at the top level.

Usage

```
ptr_shared_panel(obj, css = NULL)
```

Arguments

obj	A ptr_shared_spec from ptr_shared().
css	Optional character vector of paths to additional CSS files; linked after ggpaintr's bundled stylesheet so its rules win. See ptr_app() for the full semantics. Defaults to NULL.

Details

Namespacing is inherited from obj$id; supply it to ptr_shared().

Value

A shiny.tag div.ptr-app wrapping the wellPanel and the asset bundle, suitable for direct placement in the embedder's UI, or NULL when no panel keys exist.

See Also

ptr_shared(), ptr_ui_shared_panel(), ptr_shared_server().

Examples

```
obj <- ptr_shared(c(
  "ggplot(mtcars, aes(x = ppVar(shared='x'), y = ppVar)) + geom_point()",
  "ggplot(mtcars, aes(x = ppVar(shared='x'), y = ppVar)) + geom_bar()"
))
ptr_shared_panel(obj)
```

---

ptr_shared_server *Server-Side Counterpart to the Shared Coordinator*

Description

Builds the shared input reactives and binds the host-level ppVar(shared = "...") consumer pickers for the ptr_shared_panel(). Returns a ptr_shared_state that the embedder threads into each ptr_server() via the shared_state argument.

Usage

```
ptr_shared_server(
  obj,
  envir = parent.frame(),
  shared = list(),
  draw_trigger = NULL,
  spec = NULL
)
```

Arguments

obj	A ptr_shared_spec from ptr_shared() – the single source of truth for the cross-formula partition. Replaces the old formulas/expr_check arguments, which are now baked into obj.
envir	Environment used to resolve symbols in the shared ppVar() upstream chains. Default parent.frame() picks up the embedder's caller scope so mtcars etc. resolve naturally.
shared	Optional named list overriding the auto-derived reactives. Each named entry replaces the reactive for that key; unsupplied keys keep the default that reads input[[canonical_shared_id(key)]].

draw_trigger — Optional reactive overriding the auto-derived "Draw all" trigger (input$ptr_shared_draw_all). Useful when an embedder wants to drive the cross-module redraw from their own button.

spec — Optional named list of fully-qualified Shiny input id -> value, used to override widget defaults at session boot for host-owned panel-shared widgets (shared_<k> for consumer / value placeholders, shared_<k>_name for panel-owned source companion textInputs). Multi-instance entrypoints like ptr_app_grid() forward the same flat spec to every per-plot ptr_server() AND to this host-scope server; per-plot servers prefix-filter by their own module namespace and drop un-namespaced ids, leaving this host apply path to claim them.

Details

Namespacing is inherited from obj$id; supply it to ptr_shared().

Reads its session via shiny::getDefaultReactiveDomain() – call it inside the top-level Shiny server function (or any reactive context that inherits the session). Errors when called outside any reactive domain.

Value

A ptr_shared_state S3 object with public fields shared, draw_trigger, shared_resolutions, and shared_stage_enabled (a named list of reactives, one per shared key, indicating whether that key's shared stage is active for each embedded module).

See Also

ptr_shared(), ptr_shared_panel(), ptr_server().

Examples

```
if (interactive()) {
  obj <- ptr_shared(c(
    "ggplot(mtcars, aes(x = ppVar(shared='x'), y = ppVar)) + geom_point()",
    "ggplot(mtcars, aes(x = ppVar(shared='x'), y = ppVar)) + geom_bar()"
  ))
  shiny::shinyApp(
    ui = shiny::fluidPage(ptr_shared_panel(obj)),
    server = function(input, output, session) {
      ptr_shared_server(obj)
    }
  )
}
```

`ptr_signal_partial` *Signal a transient "partial input" failure from a placeholder hook*

Description

Placeholder `resolve_expr` (and `validate_input`) hooks should call `ptr_signal_partial()` – instead of `rlang::abort()` / `stop()` – when the current widget value is incomplete or transiently unparseable. Typical example: a free-text `expr` placeholder whose body has not finished being typed (`mpg /`). The condition carries class `"ptr_partial_input"`.

Usage

```
ptr_signal_partial(message, ...)
```

Arguments

`message`	Diagnostic text. Surfaced on the gated plot path.
`...`	Additional named fields attached to the condition (forwarded to `rlang::abort()`).

Details

ggpaintr's *live* reactive boundaries (column-picker `entry_reactives` that re-fire on every keystroke) catch this class and silently cancel the current re-render, so the downstream picker keeps its previous state instead of strobing blank and writing a Shiny warning to the R console. The *Update / Draw*-gated plot path does not catch it, so a value that is still partial when the user clicks "Update" still surfaces as a normal inline error.

Use ordinary `rlang::abort()` for failures that are NOT user mid-typing artifacts (security violations, real argument-shape errors, etc.).

Value

Never returns – always signals.

Examples

```
# A resolve hook that treats an unfinished expression as a transient,
# silently-cancelled partial input rather than a hard error:
my_expr_resolve <- function(value, node, ...) {
  tryCatch(
    rlang::parse_expr(value),
    error = function(e) ptr_signal_partial(conditionMessage(e))
  )
}
```

ptr_ui *Self-contained UI for a* ggpaintr *Formula*

Description

The L2 default-layout UI bundle for a ggpaintr formula: owns its own .ptr-app theme scope + asset bundle (nothing else to remember) and is namespaced by id. Pair with the single public ptr_server(). For a hand-composed L3 layout, use the bare ptr_ui_* pieces instead and pair them with the same ptr_server().

Usage

```
ptr_ui(
  formula,
  id = NULL,
  envir = parent.frame(),
  ui_text = NULL,
  expr_check = TRUE,
  css = NULL,
  shared = NULL
)
```

Arguments

formula	Either a single character scalar containing a ggplot expression with ggpaintr placeholders, or an unquoted ggplot expression supplied directly (the primary form). Captured with rlang::enexpr() exactly as ptr_app() / ptr_server(), so a formula stored in a variable via rlang::expr() can be passed by its bare symbol (f <- rlang::expr(ggplot(...)); ptr_ui(f, "id")) or spliced with !! (ptr_ui(!!f, "id")); a subscripted/extracted element resolves bare too (ptr_ui(plots[[1]], "id"), holder$body). See ptr_app() for the full contract (symbol resolution, wrapper unwrap, native-pipe caveat).
id	Optional module id; the namespace prefix for inputs and outputs. Defaults to NULL (identity namespace, single-instance use).
envir	Environment used to resolve a formula passed as a bare symbol and any local data objects. Defaults to the calling frame.
ui_text	Optional named list of copy overrides; see ptr_ui_text() for the full schema and current defaults.
expr_check	Controls formula-level ppExpr validation: TRUE (default) applies the built-in denylist + AST walker; FALSE disables formula-level validation; a list with deny_list/allow_list entries customises the formula-level policy. Runtime-typed ppExpr input is always screened against the built-in denylist regardless. See the safety chapter of the ggpaintr book (development-version docs): https://willju-wangqian.github.io/ggpaintr-book/safety.html.

css	Optional character vector of paths to additional CSS files; linked after ggpaintr's bundled stylesheet so its rules win. See ptr_app() for the full semantics. Defaults to NULL.
shared	Optional coordinator object from ptr_shared() for the multi-instance embedding. Forwarded verbatim to ptr_ui_controls(). When NULL (the single-instance default) the inline "Shared controls" section renders **every** shared = "..." placeholder in formula. When a ptr_shared_spec is supplied, its cross-formula keys (shared$panel_keys) are excluded here because they belong to the one standalone ptr_shared_panel(); only this formula's formula-local shared keys render inline. Defaults to NULL.

Value

A shiny.tag — a fluidPage shell containing the controls panel, plot output, and asset bundle.

See Also

ptr_server(), ptr_css() for the css = argument and themable CSS custom properties.

Examples

```
# Expression form (primary): an unquoted ggplot call.
ui <- ptr_ui(ggplot(mtcars, aes(x = ppVar, y = ppVar)) + geom_point(), "plot1")
# Stored in a variable, spliced with `!!` (paired with ptr_server(!!f, "plot1")).
f <- rlang::expr(ggplot(mtcars, aes(x = ppVar, y = ppVar)) + geom_point())
ui2 <- ptr_ui(!!f, "plot1")
# String form (fallback): equivalent.
ui3 <- ptr_ui("ggplot(mtcars, aes(x = ppVar, y = ppVar)) + geom_point()", "plot1")
```

---

ptr_ui_assets *Bundled CSS / JS Assets Piece for* ggpaintr

Description

The full ggpaintr asset bundle as a shiny::tagList(): the structural-layer dependency (ptr_set_class handler + stage CSS), the cosmetic ggpaintr.css theme dependency + code-window JavaScript, and any user override stylesheets. The CSS/JS ship as htmltools::htmlDependency() objects, so emitting this anywhere on a page — even several times — yields exactly one <head> injection of each. The single-piece UI builders (ptr_ui_plot(), ptr_ui_controls(), ...) emit **no** assets; an L3 page that composes pieces by hand normally gets them from the page shell, or includes this directly for a non-fluidPage root. The bundled apps and the ptr_*_ui() composites inject it for you.

Usage

```
ptr_ui_assets(css = NULL)
```

Arguments

css	Optional character vector of paths to additional CSS files; linked after ggpaintr's bundled stylesheet so its rules win. See ptr_app() for the full semantics. Defaults to NULL.

Value

A shiny::tagList().

See Also

ptr_ui_plot(), ptr_ui_controls(), ptr_ui_toggle_code(), ptr_css()

Examples

```
ptr_ui_assets()
```

---

ptr_ui_code *Generated-Code Pane Piece for a* ggpaintr *Formula*

Description

The generated-code output on its own: a shiny::verbatimTextOutput() bound to the ptr_code id the server writes to (see ptr_server()). One of the single-piece UI builders for the L3 "own every UI piece" workflow.

Usage

```
ptr_ui_code(id = NULL, style = c("panel", "window"))
```

Arguments

id	Optional module id; the namespace prefix for the output. Defaults to NULL (identity namespace). When set, must match the id passed to the other piece functions and the server wiring.
style	"panel" (default) renders a plain, always-visible code card suitable for free placement. "window" renders the draggable slide-out window (with Copy / Close) used by the bundled apps; it is hidden until toggled and only works when wired via ptr_ui_toggle_code().

Value

A shiny::tag.

See Also

ptr_ui_toggle_code(), ptr_ui_plot(), ptr_server()

Examples

```
ptr_ui_code("myplot")
ptr_ui_code("myplot", style = "window")
```

`ptr_ui_controls` *Controls Piece for a* `ggpaintr` *Formula*

Description

The generated control widgets (layer picker, per-layer parameter panels, the "Update plot" button) as a bare `shiny::tagList()` with **no** `.ptr-app` wrapper and **no** bundled assets. One of the single-piece UI builders for the L3 "own every UI piece" workflow: compose it with `ptr_ui_assets()` and the output pieces, place each wherever you like, and wire the server with `ptr_server()` / `ptr_server()`.

Usage

```
ptr_ui_controls(
  formula,
  id = NULL,
  envir = parent.frame(),
  ui_text = NULL,
  expr_check = TRUE,
  shared = NULL
)
```

Arguments

`formula`	Either a single character scalar containing a ggplot expression with `ggpaintr` placeholders, or an unquoted / `!!`-spliced ggplot expression, captured with `rlang::enexpr()` exactly as `ptr_app()` / `ptr_server()`. See `ptr_app()` for the full contract.
`id`	Optional module id; the namespace prefix for inputs. Defaults to `NULL` (identity namespace). When set, must match the `id` passed to the other piece functions and the server wiring.
`envir`	Environment used to resolve a `formula` passed as a bare symbol and any local data objects. Defaults to the calling frame.
`ui_text`	Optional named list of copy overrides; see `ptr_ui_text()` for the full schema and current defaults.
`expr_check`	Controls formula-level `ppExpr` validation: `TRUE` (default) applies the built-in denylist + AST walker; `FALSE` disables formula-level validation; a `list` with `deny_list`/`allow_list` entries customises the formula-level policy. Runtime-typed `ppExpr` input is always screened against the built-in denylist regardless. See the safety chapter of the ggpaintr book (development-version docs): https://willju-wangqian.github.io/ggpaintr-book/safety.html.

shared — Optional coordinator object from ptr_shared() for the multi-instance embedding. When NULL (the single-instance default) the inline "Shared controls" section renders **every** shared = "..." placeholder in formula. When a ptr_shared_spec is supplied, its cross-formula keys (shared$panel_keys) are excluded here because they belong to the one standalone ptr_shared_panel(); only this formula's formula-local shared keys render inline.

Details

Because the panel includes a shinyWidgets::pickerInput() (the layer selector) and the Bootstrap grid, it must be rendered inside a Bootstrap page that also carries the .ptr-app theme scope and the asset bundle. Don't assemble that scaffolding by hand: wrap your composed pieces in ptr_ui_page(), which *is* the Bootstrap page and owns the single .ptr-app scope + the (deduped) assets. For a navbarPage or bslib root (which ptr_ui_page() does not cover) see vignette("ggpaintr-tutorial").

For finer control still — placing individual placeholder widgets independently rather than the whole panel — register a custom placeholder type; see ptr_define_placeholder_value().

Value

A shiny::tagList().

See Also

ptr_ui_page(), ptr_ui_assets(), ptr_ui_plot(), ptr_ui_code(), ptr_shared(), ptr_server()

Examples

```
# Expression form (primary): an unquoted ggplot call.
ptr_ui_controls(
  ggplot(mtcars, aes(x = ppVar, y = ppVar)) + geom_point(),
  id = "p"
)
# String form (fallback): equivalent.
ptr_ui_controls(
  "ggplot(mtcars, aes(x = ppVar, y = ppVar)) + geom_point()",
  id = "p"
)
```

---

ptr_ui_error *Inline Error Pane Piece for a* ggpaintr *Formula*

Description

The inline error slot on its own: a shiny::uiOutput() bound to the ptr_error id the server writes parse/runtime error alerts to (see ptr_server()). One of the single-piece UI builders for the L3 "own every UI piece" workflow.

Usage

```
ptr_ui_error(id = NULL)
```

Arguments

id	Optional module id; the namespace prefix for the output. Defaults to NULL (identity namespace). When set, must match the id passed to the other piece functions and the server wiring.

Value

A shiny::tag.

See Also

ptr_ui_plot(), ptr_ui_code(), ptr_ui_controls(), ptr_server()

Examples

```
ptr_ui_error("myplot")
```

---

ptr_ui_header *App Header Piece for* ggpaintr

Description

The slim branded header bar (logo + title) the polished default shell uses in place of shiny::titlePanel(). One of the single-piece UI builders for the L3 "own every UI piece" workflow.

Usage

```
ptr_ui_header(title = "ggpaintr")
```

Arguments

title	Heading text. Defaults to "ggpaintr".

Value

A shiny::tag.

See Also

ptr_ui_controls(), ptr_ui_plot(), ptr_app()

Examples

```
ptr_ui_header()
ptr_ui_header("My App")
```

`ptr_ui_inline_error` *Nest an Inline Error Slot in a Plot Piece*

Description

Output combinator: takes an already-built bare plot piece (`ptr_ui_plot()`) and an already-built bare error piece (`ptr_ui_error()`) and returns the plot card with the error slot rendered **inline in the card body** — the layout the bundled apps use. Pure DOM structure; no server coupling (the server registers `ptr_plot` / `ptr_error` regardless). Nestable inside `ptr_ui_toggle_code()`.

Usage

```
ptr_ui_inline_error(plot, error)
```

Arguments

`plot`	A plot piece, typically `ptr_ui_plot(id)`. Must be the `.ptr-card--plot` card so the error can be appended to its body.
`error`	An error piece, typically `ptr_ui_error(id)` built with the same `id` as `plot`.

Details

This combinator does **not** add the `.ptr-output` toggle scope (it has no toggle, so it needs none); `ptr_ui_toggle_code()` owns that wrapper.

Value

A shiny::tag — the plot card with `error` nested in its body.

See Also

`ptr_ui_plot()`, `ptr_ui_error()`, `ptr_ui_toggle_code()`

Examples

```
ptr_ui_inline_error(ptr_ui_plot("p"), ptr_ui_error("p"))
```

`ptr_ui_page` *Page shell for hand-composed ggpaintr UIs*

Description

Wraps composed L3 pieces in a Bootstrap page + the single `.ptr-app` theme scope + the (deduped) asset bundle. The only thing an L3 user must remember.

Usage

```
ptr_ui_page(..., page = shiny::fluidPage, css = NULL)
```

Arguments

`...`	UI children (pieces, layout, your own widgets).
`page`	A Bootstrap-3 page builder whose `...` are tag children: `shiny::fluidPage` (default), `fixedPage`, `fillPage`, `bootstrapPage`, `basicPage`. NOT `navbarPage` (needs a positional `title` + `tabPanel` children) and NOT bslib/BS5 pages (the bundled CSS is Bootstrap-3-scoped – see `ptr_app_bslib()`). For those roots, compose by hand: see `vignette("ggpaintr-tutorial")`.
`css`	Optional character vector of extra stylesheet paths, linked after `ggpaintr.css`. See `ptr_css()`.

Value

A `shiny.tag` — the Bootstrap page node ready to pass to `shiny::shinyApp()` as `ui`.

See Also

`ptr_ui_plot()`, `ptr_ui_controls()`, `ptr_server()`, `ptr_css()`

Examples

```
f <- rlang::expr(ggplot(mtcars, aes(x = ppVar, y = ppVar)) + geom_point())
ptr_ui_page(
  shiny::sidebarLayout(
    shiny::sidebarPanel(ptr_ui_controls(id = "p", formula = !!f)),
    shiny::mainPanel(ptr_ui_plot("p"))
  )
)
```

ptr_ui_plot *Plot Pane Piece for a* ggpaintr *Formula*

Description

The plot card on its own: a `shiny::plotOutput()` bound to the `ptr_plot` id the server writes to (see `ptr_server()`). One of the single-piece UI builders for the L3 "own every UI piece" workflow; place it anywhere in your own layout and wire the server with `ptr_server()` / `ptr_server()`.

Usage

```
ptr_ui_plot(id = NULL)
```

Arguments

`id`	Optional module id; the namespace prefix for the output. Defaults to `NULL` (identity namespace) for the single-embedding case. When set, must match the `id` passed to the other piece functions and to the `shiny::moduleServer()` wrapping `ptr_server()` (or to `ptr_server()`).

Details

The piece is **truly bare**: just the plot card, with no error slot and no show-code button. Behaviour is added compositionally by the combinators `ptr_ui_inline_error()` (nests an error slot in the card body) and `ptr_ui_toggle_code()` (adds the </> toggle + slide-out code window) — not by flags on this function.

Value

A shiny::tag.

See Also

`ptr_ui_error()`, `ptr_ui_code()`, `ptr_ui_inline_error()`, `ptr_ui_toggle_code()`, `ptr_ui_controls()`, `ptr_server()`

Examples

```
ptr_ui_plot("myplot")
```

`ptr_ui_shared_panel` *Render the Standalone Shared Panel (L3, Bare)*

Description

The bare counterpart to `ptr_shared_panel()`: identical inner markup (the wellPanel holding `obj$panel_keys`) with **no** `.ptr-app` shell and **no** asset bundle. The L3 user supplies their own shell / assets (e.g. via `ptr_ui_page()`).

Usage

```
ptr_ui_shared_panel(obj)
```

Arguments

`obj`	A `ptr_shared_spec` from `ptr_shared()`.

Details

Namespacing is inherited from `obj$id`; supply it to `ptr_shared()`.

Value

A `shiny.tag` wellPanel with no wrapper and no injected assets, or `NULL` when no panel keys exist.

See Also

`ptr_shared()`, `ptr_shared_panel()`, `ptr_shared_server()`.

Examples

```
obj <- ptr_shared(c(
  "ggplot(mtcars, aes(x = ppVar(shared='x'), y = ppVar)) + geom_point()",
  "ggplot(mtcars, aes(x = ppVar(shared='x'), y = ppVar)) + geom_bar()"
))
ptr_ui_shared_panel(obj)
```

ptr_ui_text *Inspect, validate, and pre-merge ggpaintr UI copy*

Description

Returns the effective copy tree ggpaintr uses to label every generated control. Call it with no arguments to see the current defaults, or pass `ui_text =` a list of overrides to get back a validated, merged `ptr_ui_text` object. That object can be reused across `ptr_app()` / `ptr_server()` calls — those entry points short-circuit when handed an already-merged `ptr_ui_text`, so the overrides are validated once.

Usage

```
ptr_ui_text(ui_text = NULL)
```

Arguments

`ui_text`	`NULL` (return defaults), a named list of overrides, or an already-merged `ptr_ui_text` object (returned unchanged).

Details

Use it to (1) discover the override schema (`names(ptr_ui_text())` and the section below), and (2) fail fast on a malformed override list before launching an app — `ptr_ui_text()` raises on unknown sections, keywords, or leaf fields.

Value

A `ptr_ui_text` object containing the merged copy rules.

UI text schema

Override lists mirror the structure of `ptr_ui_text()`. Recognised paths (every leaf is a named list of the fields below):

- `shell$title$<leaf>`
- `shell$draw_button$<leaf>`
- `shell$draw_all_button$<leaf>`
- `shell$layer_picker$<leaf>`
- `shell$data_subtab$<leaf>`
- `shell$controls_subtab$<leaf>`
- `upload$file$<leaf>`
- `upload$name$<leaf>`
- `layer_checkbox$<leaf>`
- `defaults$<keyword>$<leaf>` — per placeholder keyword (`ppVar`, `ppText`, `ppNum`, `ppExpr`, `ppUpload`, ...)

- `params$<param>$<keyword>$<leaf>` — per aesthetic/argument name (`x`, `y`, `color`, ...); aliases (`colour`, `size`) are normalized
- `layers$<layer_name>$<keyword>$<param>$<leaf>` — per layer override (use `__unnamed__` as `<param>` for positional arguments)

Leaf fields are `label`, `help`, `placeholder`, and `empty_text`. Leaf strings may use the `{param}` and `{layer}` tokens, which are interpolated at resolve time.

Examples

```
# Default rules
rules <- ptr_ui_text()
rules$shell$title$label

# Override the draw button label
rules <- ptr_ui_text(
  ui_text = list(shell = list(draw_button = list(label = "Render")))
)
rules$shell$draw_button$label
```

`ptr_ui_toggle_code` *Wire a Plot-ish Piece to a Slide-Out Code Window via the </> Toggle*

Description

Output combinator: wraps a plot-ish tag (a bare `ptr_ui_plot()` or the output of `ptr_ui_inline_error()`) and a bare code piece (`ptr_ui_code()`) in the single `.ptr-output` scope the bundled JavaScript needs, injecting the </> show-code button into the plot card head and presenting the code inside the draggable slide-out `.ptr-code-window` (with Copy / Close). The button hides/shows that window purely DOM-locally — no Shiny input/output is involved. This is the toggle layout `ptr_app()` / `ptr_ui()` render internally.

Usage

```
ptr_ui_toggle_code(plotish, code)
```

Arguments

`plotish`	A plot-ish tag. The designed input is a bare `ptr_ui_plot(id)` or `ptr_ui_inline_error(ptr_ui_plot(id), ptr_ui_error(id))` — a single `.ptr-card--plot` tag, where the toggle button is injected into the card head (DOM byte-identical to the bundled output block). An arbitrary custom output also works: a `shiny.tag.list` (e.g. `plotly::plotlyOutput()`, `ggiraph::girafeOutput()`) or a childless single tag (e.g. `shiny::plotOutput()`) gets the toggle button as an explicit sibling inside the `.ptr-output` wrapper instead.
`code`	A code piece, typically `ptr_ui_code(id)` built with the same `id`. Its style is irrelevant — this combinator supplies the slide-out window chrome around it.

Details

Use this when you want the familiar toggle behaviour while still owning the surrounding layout. For fully independent placement of the plot and code panes (no toggle), keep the bare pieces uncombined — a standalone `ptr_ui_code()` is always visible and needs no wiring.

Value

A shiny::tag — one `.ptr-output` containing `plotish` (with the toggle button) and the `.ptr-code-window`-wrapped `code`.

See Also

`ptr_ui_plot()`, `ptr_ui_code()`, `ptr_ui_inline_error()`, `ptr_ui_controls()`, `ptr_server()`

Examples

```
ptr_ui_toggle_code(
  ptr_ui_inline_error(ptr_ui_plot("p"), ptr_ui_error("p")),
  ptr_ui_code("p", style = "window")
)
```

---

ptr_wrap_placeholder_addin

RStudio addin: wrap a selection in a ggpaintr placeholder

Description

Interactive RStudio addin. Highlight a token in your ggplot expression (e.g. `mpg` in `aes(x = mpg)`), run the addin, and pick a placeholder from a command-palette gadget; the selection is rewritten to `ppVar(mpg)`. With nothing highlighted the same palette opens and inserts `ppVar()` with the caret between the parens. The placeholder list is read live from the registry, so custom placeholders registered this session (via `ptr_define_placeholder_value()` and friends) appear automatically.

Usage

```
ptr_wrap_placeholder_addin()
```

Details

The gadget's *Wrap in app* button (left of *Insert*) takes a different action: instead of inserting a placeholder it wraps the whole selection in a braced block piped into `ptr_app()` – `{ / <selection> / } |> / ptr_app()` – turning a ggplot expression into a runnable ggpaintr app skeleton.

Value

Invisibly `NULL`. Called for its side effect of editing the active RStudio document.

Theme

By default the palette follows your RStudio editor theme (dark theme -> dark palette, light -> light), via `rstudioapi::getThemeInfo()`. Force one with `options(ggpaintr.addin_theme = "dark")`, `"light"`, or `"auto"` (the default).

Keyboard shortcut

For a highlight-then-keystroke flow, bind the addin once (RStudio reads shortcuts only from your own keybindings, so packages cannot ship one). The addin must be **installed** (not merely `load_all()`-ed) to appear in the shortcut dialog:

1. *Tools* > *Addins* > *Browse Addins...*, then the *Keyboard Shortcuts...* button. (Or *Tools* > *Modify Keyboard Shortcuts...* and type "ggpaintr" in the search box.)
2. Find the *ggpaintr placeholder* row and click its *Shortcut* cell.
3. Press the recommended combination: **Cmd+Shift+G** on macOS, **Ctrl+Shift+G** on Windows/Linux. (Any free combination works; pick another if that one is already bound.)
4. *Apply*.

Requires **rstudioapi** and **miniUI**; it errors with a clear message when run outside RStudio.

Index