# Package: blockr.dock (via r-universe)

November 20, 2026

**Title** A Docking Layout Manager for 'blockr'

**Version** 0.2.0

**Description** Building on the docking layout manager provided by 'dockViewR', this provides a flexible front-end to 'blockr.core'. It provides an extension mechanism which allows for providing means to manipulate a board object via panel-based user interface components.

**URL** https://bristolmyerssquibb.github.io/blockr.dock/

**BugReports** https://github.com/BristolMyersSquibb/blockr.dock/issues

**License** GPL (>= 3)

**Encoding** UTF-8

**Roxygen** list(markdown = TRUE)

**Imports** blockr.core (>= 0.1.4), blockr.ui, bsicons, shiny (>= 1.14.0), shinyjs, bslib, dockViewR (>= 0.4.0), htmltools, cli, jsonlite, rlang, utils, reactives

**Suggests** testthat (>= 3.0.0), DT, colorspace, methods, withr, shinytest2, chromote, xml2, knitr, rmarkdown

**Remotes** rstudio/shinytest2, BristolMyersSquibb/blockr.core, BristolMyersSquibb/blockr.ui, cynkra/dockViewR, nbenn/reactives

**Config/testthat/edition** 3

**Collate** 'action-class.R' 'action-block.R' 'action-inputs.R' 'action-link.R' 'action-menu.R' 'action-stack.R' 'action-ui.R' 'action-utils.R' 'block-meta.R' 'block-ui.R' 'board-options.R' 'board-plugins.R' 'board-server.R' 'board-ui.R' 'dock-board.R' 'dock-grid.R' 'dock-layout.R' 'dock-navbar.R' 'dock-rail.R' 'dock-stack.R' 'dock-view.R' 'ext-class.R' 'ext-delta.R' 'ext-edit.R' 'ext-meta.R' 'ext-ui.R' 'panel-id.R' 'panel-menu.R' 'panel-ops.R' 'panel-ref.R' 'plugin-block.R' 'plugin-code.R' 'plugin-serdes.R' 'sidebar-server.R' 'sidebar-block.R' 'sidebar-inputs.R' 'sidebar-link.R' 'sidebar-stack.R' 'utils-dock.R' 'utils-misc.R' 'utils-pkg.R' 'utils-serdes.R' 'utils-serve.R' 'utils-ui.R'

**VignetteBuilder** knitr

**Config/roxygen2/version** 8.0.0

**Config/pak/sysreqs** cmake make libuv1-dev zlib1g-dev

**Repository** https://cynkra.r-universe.dev

**Date/Publication** 2026-11-20 15:20:49 UTC

**RemoteUrl** https://github.com/BristolMyersSquibb/blockr.dock

**RemoteRef** HEAD

**RemoteSha** d8b30e148a43612e580f8970ceaf85b89e7fb6d2

# Contents

blk . . . 2
blks_metadata . . . 4
dock-grid . . . 6
dock-layout . . . 6
dock_grid . . . 7
dock_id . . . 9
ext-meta . . . 11
is_dock_grids . . . 13
navbar_item . . . 15
new_action . . . 17
new_compact_option . . . 19
new_dock_board . . . 20
new_dock_extension . . . 22
new_dock_stack . . . 24
new_edit_board_extension . . . 25
new_sync_rails_option . . . 26
new_view_tabs_option . . . 27
panel_obj_ids . . . 27

**Index** **29**

---

`blk` *Typed panel references*

---

## Description

`blk()` and `ext()` construct references to a block or extension panel for the `views$mod` panel-op grammar (see board_update). They are the public currency: a caller names a block or extension by its **id**, never the `block_panel-` / `ext_panel-` wire prefix – the ref is the codec, and `as.character()` on one yields the canonical panel-id encoding.

## Usage

```
blk(id, near = NULL, side = NULL, size = NULL, rail = NULL)

ext(id, near = NULL, side = NULL, size = NULL, rail = NULL)

is_panel_ref(x)

as_panel_ref(
  id,
  block_ids = character(),
  ext_ids = character(),
  near = NULL,
  side = NULL,
  size = NULL,
  rail = NULL
)

## S3 method for class 'panel_ref'
as.character(x, ...)
```

## Arguments

| | |
|---|---|
| `id` | A block or extension id. For `blk()` / `ext()` a bare id (not the wire-prefixed panel id); `as_panel_ref()` additionally accepts an already wire-prefixed panel id or a `panel_ref`. |
| `near` | A ref or bare id to anchor placement against. |
| `side` | Placement direction relative to `near`: one of `within`, `left`, `right`, `above`, `below`. |
| `size` | Target size ratio in (0, 1) – consumed by `resize`, and recorded on `add` for a later size-on-create pass. |
| `rail` | The edge to park the panel on, `left` or `right`. Mutually exclusive with `near` / `side`. |
| `x` | An object. |
| `block_ids`, `ext_ids` | Character vectors partitioning the id namespace that `as_panel_ref()` resolves a bare `id` against. |
| `...` | Ignored. |

## Details

Each ref optionally carries its own placement hint, so a verb's operands are an unnamed list of self-describing refs: `add = list(blk("a", near = "b", side = "right"), ext("dag"))`. Which hint fields are meaningful depends on the verb – `add` consumes `near` / `side` / `size` / `rail`, `move` consumes `near` / `side` / `rail` – and a hint on a ref used where no placement happens (`rm`, `select`, a `near` anchor, or the `dock_grid()` / `panels()` authoring DSL) is a loud error. Because the hints are constructor arguments, a misspelled one (`blk("a", sise = 0.4)`) fails at the call site with R's own unused-argument error, before any payload exists.

The rail hint parks a panel on one of the view's edges (see rail()), which is the one destination the tree cannot express, so it excludes near and side. Keeping the two apart matters beyond the pairing: side is a direction relative to a near anchor *inside* the splitview, a rail position is an edge of the whole view, and both spell left and right.

Bare id strings are accepted as sugar wherever a ref is, resolved block-first with a hard error only on a true cross-namespace clash (an id that is both a block and an extension), which then demands a typed ref.

as_panel_ref() is the programmatic form of that sugar: given a bare id and the block_ids / ext_ids partitioning a board's namespace, it returns the correctly-typed blk() / ext() ref (carrying any placement hint), applying the same block-first precedence and the same cross-namespace clash error. An already wire-prefixed panel id, or an existing panel_ref, resolves to the matching ref unchanged.

## Value

blk(), ext() and as_panel_ref() return a panel_ref. as.character() on one returns its canonical panel id, and is_panel_ref() returns a boolean.

## Examples

```
blk("my_block", near = "other_block", side = "right")
ext("dag", rail = "left")
as.character(blk("my_block"))

as_panel_ref("my_block", block_ids = "my_block")
as_panel_ref("dag", block_ids = "my_block", ext_ids = "dag")
```

---

# blks_metadata *Get block metadata*

---

## Description

Returns various metadata for blocks or block categories, as well as styling for block icons.

## Usage

```
blks_metadata(blocks)

blk_color(category)

blk_icon_data_uri(icon_svg, color, size = 48, mode = c("uri", "inline"))

block_status_badge(status, error_count = 0L)
```

## Arguments

| | |
|---|---|
| blocks | Blocks passed as blocks or block object |
| category | Block category |
| icon_svg | Character string containing the SVG icon markup |
| color | Hex color code for the background |
| size | Numeric size in pixels (default: 48) |
| mode | Switch between URI and inline HTML mode |
| status | A block eval status: stale, waiting, unset and failed carry a badge; ready and unevaluated carry none; any other value yields no badge. The size field is the coloured dot's pixel diameter and ring the width of the ring around it, both shared by the dock card icon and the DAG node badge. |
| error_count | Number of error conditions the block has raised. A positive count promotes the badge to failed, catching render-phase errors that leave the eval status ready. A stale block is exempt: its conditions were raised against inputs it no longer has. |

## Details

- blks_metadata(): Retrieves metadata given a block or blocks object from the block registry. Can also handle blocks which are not registered and provides default values in that case. The color column is the category's colour, from blockr.ui::category_color().
- blk_color(): Deprecated. The category colours are blockr.ui's, and blockr.ui::category_color(), which this now calls, looks them up.
- blk_icon_data_uri(): Deprecated. Processes block icons to add color and turn them into square-shaped icons. For the block's mark as an image, use blockr.ui::block_mark_svg(), which draws it from the block's glyph and category.
- block_status_badge(): Derives a block's status badge from its eval status and error count – the single derivation the dock card icon and the blockr.dag node badge share, so both show the same status. Returns a styling list (draw the badge) or NULL (no badge).

## Value

Metadata is returned from blks_metadata() as a data.frame with each row corresponding to a block. Both blk_color() and blk_icon_data_uri() return character vectors. The badge block_status_badge() draws is a list with its label; its fill as the blockr.ui token and that token's light value color; hollow and outline, for a badge drawn as an outline-wide ring in its fill; and the dot's size and the width of the ring around it, in ring_token with the light value ring_color. It is NULL for a status with no badge.

## Examples

```
blk <- blockr.core::new_dataset_block()
blks_metadata(blk)

block_status_badge("waiting")
```

## dock-grid *Canonical view grid*

### Description

A dock_grid is a view's geometry in our compact, dockView-independent form – nested splits and tab groups with sizes normalised to 0-1 ratios and no volatile ids, so two casts of the same layout compare identical(). It is authored with dock_grid() and produced by as_dock_grid(), which casts another dock_grid (identity) or a dock_layout (dockView's client echo) into it and is idempotent. is_dock_grid() is the class check; validate_dock_grid() returns its input and errors on a malformed or non-canonical grid, or on one that places a panel more than once.

### Usage

```
as_dock_grid(x, ...)

is_dock_grid(x)

validate_dock_grid(x)
```

### Arguments

| | |
|---|---|
| x | Object to cast (a dock_grid or a dock_layout), validate, or test. |
| ... | Passed on to methods. |

### Value

as_dock_grid() and validate_dock_grid() a dock_grid; is_dock_grid() a boolean.

## dock-layout *Dock layout: dockView's native representation*

### Description

A dock_layout is dockView's own layout representation – the {grid, panels, activeGroup} payload it renders – the wire form at the client boundary. It is distinct from our dock_grid (a view's canonical geometry) and dock_view() (a view's structure): a dock_layout is what the browser echoes and what a restore pushes back, geometry fused with resolved panel content in dockView's own shape. It is produced and consumed only at the dockView seam and is never stored on the board nor passed through the update lifecycle.

## Usage

```
new_dock_layout(x = list())

is_dock_layout(x)

validate_dock_layout(x)

as_dock_layout(x, ...)
```

## Arguments

| | |
|---|---|
| x | Object to wrap, validate, test, or cast. |
| ... | Passed on to methods (e.g. blocks / extensions to resolve panel content when casting a dock_grid). |

## Details

new_dock_layout() wraps a raw dockView state / payload list (a client echo) as a dock_layout; is_dock_layout() is the class check; validate_dock_layout() returns its input and errors on a malformed shape. Cast across the seam with as_dock_layout() (build the dockView payload from a dock_grid against the board's blocks and extensions), as_dock_grid() (canonical geometry from a layout, rails included) and as_dock_view() (membership from a layout).

## Value

new_dock_layout(), validate_dock_layout() and as_dock_layout() return a dock_layout; is_dock_layout() a boolean.

---

# dock_grid *Dock grid authoring*

---

## Description

A dock_grid is a view's geometry: the arrangement of its panels into nested splits and tab groups, with sizes. dock_grid() is the authoring DSL that builds one; view *membership* (which panels belong) and the display *name* live on the view (see dock_view()), not the grid.

## Usage

```
dock_grid(..., orientation = c("horizontal", "vertical"), sizes = NULL)

panels(..., active = NULL)

group(..., sizes = NULL)

default_layout(blocks, extensions)
```

```
rail(
  ...,
  position = c("left", "right"),
  active = NULL,
  collapsed = FALSE,
  size = 260,
  collapsed_size = 35
)
```

## Arguments

| | |
|---|---|
| `...` | For `dock_grid()` and `group()`, grid children (`blk()` / `ext()` references, bare ids, character vectors, lists, `panels()`, or `group()`). For `panels()`, panel references or ids. A reference used here cannot carry a placement hint. Otherwise reserved for generic consistency. |
| `orientation` | Top-level split direction; one of `"horizontal"` (default) or `"vertical"`. |
| `sizes` | Numeric vector parallel to `...`, giving each child's share of the parent (positive; need not sum to 1). |
| `active` | For `panels()` and `rail()`, the id of the tab to open by default. |
| `blocks, extensions` | Dock board components to arrange (for `default_layout()`). |
| `position` | For `rail()`, the edge the rail pins to; `"left"` (the default) or `"right"`. |
| `collapsed` | For `rail()`, whether it opens collapsed to its bare tab strip. A user collapses a rail by clicking its open tab, and that choice is stored, so a restored board comes back the way it was left. |
| `size, collapsed_size` | For `rail()`, its width (or height, on a horizontal edge) in pixels when open and when collapsed to its bare tab strip. |

## Details

Construct a grid with:

- `dock_grid(...)`: the view-level container. Its `...` are the children of the root branch. Bare strings become single-panel leaves, character vectors become tabbed leaves, lists become nested branches. Use `panels()` for a tabbed leaf with an explicit open tab, and `group()` for a branch with explicit sizes.
- `panels(..., active = NULL)`: a tabbed leaf whose tab strip holds the given panel ids. `active` selects the initially-open tab; the first id wins by default. A single-panel `panels()` is permitted but redundant (a bare string is equivalent).
- `group(..., sizes = NULL)`: a branch container. `sizes` is a numeric vector parallel to `...` that overrides the even split.
- `rail(..., position = "left")`: a tab group pinned to one edge of the view rather than arranged by the splitview. Write it among the grid's children; it is not one, so it never counts towards `sizes`, and the members it names leave the grid tree.

- `default_layout(blocks, extensions)` produces the default board arrangement (an extension rail on the left, blocks tabbed in the grid) as a `list(views, grids)` the constructor consumes.

`dock_grid()` accepts `orientation = "horizontal" | "vertical"` for the top-level split direction and `sizes` for the root-branch ratios. The dockView-native `{grid, panels, activeGroup}` payload dockViewR consumes is a dock_layout, built from a grid against the board's blocks and extensions.

## Value

`dock_grid()` returns a dock_grid object. `panels()` returns a `dock_panels` node and `group()` returns a `dock_group` node – both are grid sub-trees usable inside `dock_grid()` / `group()`. A `rail()` is a `dock_rail`, written among a grid's children but pinned outside the splitview. Finally `default_layout()` returns a `list` with `views` (a `dock_views`) and `grids` (a `dock_grids`).

## Examples

```
blks <- c(
  a = blockr.core::new_dataset_block(),
  b = blockr.core::new_head_block()
)

# Panels named with typed references; bare id strings work too
panels(blk("a"), blk("b"), active = blk("b"))

# Branch with explicit child ratios
group(blk("a"), blk("b"), sizes = c(0.3, 0.7))

# An extension panel beside a tabbed block leaf
dock_grid(
  ext("dag"),
  panels(blk("a"), blk("b"), active = blk("b")),
  sizes = c(0.3, 0.7)
)

# Vertical top-level split
dock_grid(blk("a"), blk("b"), orientation = "vertical")

# An extension parked on the left edge, out of the splitview
dock_grid(blk("a"), rail(ext("dag"), position = "left"))
```

---

`dock_id` *ID utilities*

---

## Description

Objects, such as blocks and dock extensions carry their own IDs. These can be converted into other ID types, such as panel IDs or "handle" IDs. Panel IDs are used to refer to dock panels, while handle IDs provide "handles" for DOM manipulations. All such IDs inherit from `dock_id` and panel IDs additionally inherit from `dock_panel_id`, while handle IDs inherit from `dock_handle_id`. For panel IDs, depending on whether the panel is showing a block or an extension, the inheritance structure additionally contains `block_panel_id` or `ext_panel_id`, respectively. Similarly, for handle IDs, we have `block_handle_id` and `ext_handle_id` for block / extension cards, plus `view_handle_id` for a view's DOM container. All `dock_id` objects can be converted back to native IDs, by calling `as_obj_id()`. The utility function `dock_id()` returns a (possibly namespaced) ID of the dock instance that is used to manage all visible panels.

## Usage

```
dock_id(ns = NULL)

is_dock_panel_id(x)

as_dock_panel_id(x)

as_obj_id(x)

is_block_panel_id(x)

as_block_panel_id(x)

is_ext_panel_id(x)

as_ext_panel_id(x)

as_dock_handle_id(x)

as_block_handle_id(x)

as_ext_handle_id(x)

as_view_handle_id(x)
```

## Arguments

| | |
|---|---|
| `ns` | Namespace prefix |
| `x` | Object |

## Value

Coercion functions `as_block_panel_id()`, `as_ext_panel_id()`, `as_block_handle_id()` and `as_ext_handle_id()` return objects that inherit from `block_panel_id`, `ext_panel_id`, `block_handle_id` and `ext_handle_id` as classed character vectors. The less specific coercion functions `as_dock_panel_id()`

and as_dock_handle_id() return objects that inherit from dock_panel_id and dock_handle_id, in addition to a sub-class such as block_panel_id or ext_panel_id (in the case of as_dock_panel_id()). If a mix of sub-classes is returned, this will be represented by a list of classed character vectors. as_view_handle_id() maps a view id to its DOM container id (a view_handle_id). Finally, as_obj_id() returns a character vector, as does dock_id(). The predicates is_dock_panel_id(), is_block_panel_id() and is_ext_panel_id() return a boolean, testing whether an object inherits from dock_panel_id (and, for the latter two, its block_panel_id / ext_panel_id sub-class).

## Examples

```
blks <- c(
  a = blockr.core::new_dataset_block(),
  b = blockr.core::new_head_block()
)

ext <- new_edit_board_extension()

as_dock_panel_id(blks)
as_dock_panel_id(ext)

identical(names(blks), as_obj_id(as_block_panel_id(blks)))

as_dock_handle_id(blks)
as_dock_handle_id(ext)

identical(names(blks), as_obj_id(as_block_handle_id(blks)))
```

---

# ext-meta *Extension metadata*

---

## Description

The description argument of new_dock_extension() accepts either a bare string or a structured ext_meta object built with new_ext_meta(). The structured form carries model-facing documentation for a client driving the extension through modify_extension: a free-text description, per-variable arguments documentation keyed by externally controllable variable (see external_ctrl_vars()), worked examples and free-text guidance on how to drive it. is_ext_meta() checks the class and ext_meta() reads the whole metadata off an extension, coercing a bare-string description to an ext_meta whose only populated slot is description; the individual components are read with ext_desc(), ext_args(), ext_examples() and ext_guidance(). extension_description() remains as a deprecated alias of ext_desc().

## Usage

```
new_ext_meta(
  description = NULL,
  arguments = NULL,
  examples = list(),
```

```
  guidance = NULL
)

is_ext_meta(x)

ext_meta(x)

ext_desc(x)

ext_args(x)

ext_examples(x)

ext_guidance(x)

extension_description(x)
```

## Arguments

| | |
|---|---|
| description | Free-text summary of what the extension is |
| arguments | Per-variable documentation for the externally controllable variables, either a named character vector (variable to description) or a new_arg_specs() object; keyed by controllable variable |
| examples | List of worked configurations, each a named list keyed by controllable variable (a modify_extension-shaped payload) |
| guidance | Free-text steering on how to drive the extension, distinct from the human-facing description |
| x | An ext_meta object for is_ext_meta(), a dock_extension for ext_meta() and the per-component accessors |

## Details

Per-variable arguments reuse blockr.core's block-argument specification: pass a named character vector (variable to description) for the common case, or a new_arg_specs() object to attach a machine-readable type (via the arg_*() constructors) and a worked example to each variable.

## Value

new_ext_meta() returns an ext_meta object (a list with entries description, arguments, examples and guidance), is_ext_meta() a boolean and ext_meta() the normalized ext_meta an extension carries. The per-component accessors return that component: ext_desc() a string or NULL, ext_args() an arg_specs, ext_examples() a list and ext_guidance() a string or NULL.

## Examples

```
new_ext_meta(
  "Workflow diagram of the board's blocks and links.",
  arguments = c(positions = "JSON object mapping block id to {x, y} coords.")
```

```
)

ext_meta(new_edit_board_extension())
```

---

`is_dock_grids` *Dock views: structure and grid*

---

## Description

A `dock_board` stores its views as three independent slots. **Structure** – which panels belong to each view, plus the view names, ids and the active view – is a `dock_views` collection of `dock_view` objects, read with `board_views()`. **Grid** – the geometry of each view (nesting, tab groups, sizes) – is a separate, `NULL`-valid `dock_grids` slot, read with `board_grids()`; a grid also carries the **rails** pinned to the view's edges, holding members the splitview therefore does not arrange. Single-view boards are a degenerate case: one auto-named "Page" view. Blocks and extensions are shared across views via the board's DAG; view membership is a layout concern only.

## Usage

```
is_dock_grids(x)

validate_dock_grids(x, views = NULL)

board_grids(x)

board_grids(x) <- value

as_dock_view(x, ...)

dock_view(members = character(), name = NULL)

is_dock_view(x)

validate_dock_view(x)

view_members(x)

is_dock_views(x)

validate_dock_views(x)

view_name(x)

view_name(x) <- value

view_names(x)
```

```
active_view(x)

active_view(x) <- value

board_views(x)

board_views(x) <- value
```

## Arguments

| | |
|---|---|
| x | An object appropriate to the function: a dock_view (for view_name(), view_members()), a dock_views collection (for view_names(), active_view()), a dock_grids, or a dock_board (for the board accessors). |
| views | A dock_views collection, used to check that grids key known views. |
| value | Replacement value |
| ... | Forwarded to methods. |
| members | Ordered character vector of panel ids. |
| name | Optional display name for the view. |

## Details

Each view carries a stable, immutable **id** (its key in the collection) distinct from its editable display **name**. This mirrors the id / name separation used for blocks (an immutable id keys the collection; blockr.core::block_name() is an editable label). The id is minted once when the view is created and never changes – rename only rewrites the name attribute, never the key. Use dock_view() to construct a view, view_name() / view_name<-() to read and write its display name, view_names() for all names in a collection, and view_members() for a dock_view's ordered panel-id set.

Multi-view boards are defined by passing views (and optionally grids) to new_dock_board(): views is a named list keyed by view id (minted when absent), each value a dock_view(), a bare character vector of member panel ids, or a list of panel ids. grids is a named list keyed by view id whose values are dock_grid()s; it is optional – a view with no grid entry falls back to a default grid over its members wherever placement geometry is needed. Bare block / extension ids in either slot are resolved to canonical panel ids against the board's blocks and extensions. The initially-active view is chosen by new_dock_board(active = ) (a view id), defaulting to the first; it is a property of the collection, never of an individual view. View CRUD is enabled unless the dock is locked (see is_dock_locked()).

Structure and geometry are related by total semantics, not containment: a member with no grid entry is an un-landed intent, a grid entry with no membership an inert ghost. Both are legal on a committed board and reconciled only where placement is read (view_grid() prunes ghosts and shows un-landed members via a default) – the board is valid with no grid at all. The rails partition that placement rather than adding to it: a member a rail names is placed by the rail and so is absent from the grid, and a view's membership is the union of the two. Referential integrity still holds: every member must reference a block or extension on the board.

## Value

board_views() returns a dock_views, board_grids() a dock_grids or NULL, and their setters the modified board invisibly. A dock_view() is a dock_view and view_members() a character vector. The predicates is_dock_view() / is_dock_views() / is_dock_grids() return a boolean, while validate_dock_view(), validate_dock_views() and validate_dock_grids() return their (validated) input and throw on error. active_view() returns the active view's id, or NULL when no view is active, and active_view<-() the modified collection (or dock_board) invisibly. view_name() returns a view's explicit display name (or NULL), view_name<-() the modified view, and view_names() a character vector of display labels keyed by view id (derived from the id where a view has no explicit name). as_dock_view() returns a dock_view: identity on a dock_view, or a view whose members are a dock_layout's panel ids.

## Examples

```
brd <- new_dock_board(
  blocks = c(
    dataset_1 = blockr.core::new_dataset_block(),
    head_1 = blockr.core::new_head_block()
  ),
  views = list(
    analysis = dock_view(c("dataset_1", "head_1"), name = "Analysis"),
    overview = "dataset_1"
  ),
  active = "overview"
)
view_names(board_views(brd))
view_members(board_views(brd)[["analysis"]])
```

---

# navbar_item *Navbar items*

---

## Description

The navbar of a dock board is drawn from a navbar_items collection, the dock's own controls included. An item is created by navbar_item(), items are combined with c(), [ reorders or drops them by id and [[ extracts one. An app hands the navbar to blockr.core::serve() as navbar, a function of the board and its plugins that returns the items, as it does plugins and options.

## Usage

```
navbar_item(id, ui, server = NULL, fill = FALSE, shrink = FALSE)

is_navbar_item(x)

is_navbar_items(x)

plugin_navbar_item(plugin)
```

```
default_navbar_items(x, plugins)

blockr_app_navbar(x, plugins)

custom_navbar(x)
```

## Arguments

| | |
|---|---|
| id | Item id, unique within the navbar |
| ui | A function called as ui(id, board) with id as NS(board_id, item_id) and the board, returning the item's UI, or NULL to leave the item out of the bar |
| server | NULL or a function called as server(id, board) with the item id and the board's reactive values, once per session, from the board server, so that a shiny::moduleServer() it starts shares the UI's namespace |
| fill | Whether the item takes up the free space of the bar |
| shrink | Whether the item gives way first on a narrow bar. The items with neither fill nor shrink keep their width. |
| x | Object. For default_navbar_items() and blockr_app_navbar(), the board, and for custom_navbar(), the items to append: a navbar_item, a navbar_items collection or a list of items. |
| plugin | A board plugin, or NULL |
| plugins | Board plugins |

## Details

The default items, returned by default_navbar_items(), are, in order: the blockr logo, which is also the busy indicator (id "logo"), the piece the preserve_board plugin draws ("preserve_board"), a spacer ("spacer"), the view menu ("views"), which shows the views as a line of tabs under the bar while the board's new_view_tabs_option() is on, the read-only indicator ("read_only"), drawn only on a locked board, and the board options ("options"). The generic blockr_app_navbar() gives the default by board class, and custom_navbar() returns a navbar function that appends a fixed set of items to it, dropping any default item that shares an id with one of them. A plugin's piece is placed by plugin_navbar_item(), which the defaults use for the preserve_board plugin.

An item whose ui draws a single shiny::uiOutput() or shiny::textOutput() takes no room in the bar while that output is empty, as an avatar would with nobody signed in.

## Value

The constructor navbar_item() returns a navbar_item object, and so does plugin_navbar_item(), unless its plugin is NULL, which returns NULL. The functions default_navbar_items() and blockr_app_navbar() return a navbar_items collection, as do c() and [ on items, and custom_navbar() returns a function of the board and its plugins that returns one. The checks is_navbar_item() and is_navbar_items() return a boolean.

## Examples

```
help <- navbar_item(
  "help",
  function(id, board) {
    shiny::tags$a(
      href = "https://bristolmyerssquibb.github.io/blockr.dock/",
      "Help"
    )
  }
)

brd <- new_dock_board()
items <- c(default_navbar_items(brd, blockr.core::board_plugins(brd)), help)
names(items)
names(items[c("help", "options")])

if (interactive()) {
  blockr.core::serve(brd, navbar = custom_navbar(help))
}
```

---

`new_action` *Board actions*

---

## Description

Logic including a modal-based UI for board actions such as "append block" or "edit stack" can be specified using `action` objects, which essentially are classed shiny server functions.

## Usage

```
new_action(func, id)

is_action(x)

is_action_generator(x)

action_id(x)

board_actions(x, ...)

action_triggers(x)

block_input_select(
  block = NULL,
  block_id = NULL,
  links = NULL,
  mode = c("create", "update", "inputs"),
```

```
  ...
)

board_block_select(
  id,
  board,
  blk_ids = board_block_ids(board),
  selected = NULL,
  max_items = 1L,
  label = NULL,
  options = list()
)

sidebar_owned_by(action, board_id, session = get_session())
```

## Arguments

| | |
|---|---|
| `func` | A function which will be used to create a `shiny::moduleServer()`. |
| `id` | Input ID |
| `x` | Object |
| `...` | Forwarded to other methods |
| `block` | Block object |
| `block_id` | Block ID |
| `links` | Links object |
| `mode` | Switch for determining the return object |
| `board` | Board object |
| `blk_ids` | Character vector of block IDs to offer for selection |
| `selected` | Character vector of pre-selected block IDs |
| `max_items` | Maximum number of blocks that can be selected at once, or `NULL` for no limit |
| `label` | Input label |
| `options` | Passed to `shiny::selectizeInput()` as `options`, merged over the defaults that make up the picker |
| `action` | Action ID |
| `board_id` | ID of the board module the action is registered with |
| `session` | Shiny session |

## Details

An action is a function that can be called with arguments `input`, `output` and `session`, behaving as one would expect from a shiny server module function. Actions are typically created by action generator functions, they each have a unique ID and a `shiny::reactiveVal()`-based trigger object (inheriting from `action_trigger`). Action trigger objects implement their own counter-based invalidation mechanism (on top of how reactive values behave). A trigger fires with its target, such as the ID of the block to append to, and may name where the gesture happened as `at`: a list holding

the id of an element, or the x and y of a point in the viewport. An action that opens a menu in place opens it there, and in a fixed spot when the trigger names none.

An action that fills a sidebar panel records itself on it as it writes, so the panel reports the writing action as its owner alongside whether it is open and pinned. sidebar_owned_by() reads that back for a given action: a consumer re-firing an action on a new selection can ask whether the form it previously opened is still the one on screen, without knowing which panel that action fills. The read is a snapshot and creates no reactive dependency, as called for in an shiny::observeEvent() handler.

## Value

The constructor new_action returns a classed function that inherits from action. Inheritance can be checked with functions is_action(), is_action_generator() checks whether an objects is a function that returns an action object. String-value action IDs can be retrieved with action_id() and the set of actions associated with a board can be enumerated via board_actions(). Finally, action_triggers() returns a named list of objects suitable for use as action triggers. For an action that currently holds a sidebar panel, sidebar_owned_by() returns a list with components panel, open and pinned, and NULL otherwise.

For utilities block_input_select() and board_block_select(), see the respective sections.

### block_input_select()

Determine input options for a block by removing inputs that are already used and also takes into account some edge-cases, such as variadic blocks. If mode is set as "inputs", this will return a character vector, for "create", the return value of a shiny::selectizeInput() call and for "update", the return value of a shiny::updateSelectizeInput() call.

### board_block_select()

Block selection UI over the blocks of a board is available as board_block_select(), which returns an object inheriting from shiny.tag.list: the result of a shiny::selectizeInput() call together with the styling its option rendering requires. This is the picker the board itself uses wherever a block is chosen, listing each block by icon, name, ID and defining package, and searchable over all of those.

---

# new_compact_option *Compact block headers*

---

## Description

A board option that switches every block header to its compact form: an eyebrow line over the block's content, with the block's mark as a small tinted square and the name in small capitals. Off by default, which keeps the regular header (a 32px mark and a 16px title that may wrap to two lines). It sits with the light/dark switch under "Theme options".

## Usage

```
new_compact_option(
  value = blockr_option("compact", FALSE),
  category = "Theme options",
  ...
)
```

## Arguments

| | |
|---|---|
| value | Logical, whether headers start compact. Defaults to the compact blockr option, else FALSE. |
| category | Options sidebar category. |
| ... | Passed to blockr.core::new_board_option(). |

## Value

A board_option object.

## Examples

```
new_compact_option(TRUE)
```

---

# new_dock_board *Dock board*

---

## Description

Using the docking layout manager provided by dockViewR, a dock_board extends blockr.core::new_board(). In addition to the attributes contained in a core board, this also includes dock extensions (as extensions) and the per-view layout, stored as two independent slots – view structure (board_views()) and grid geometry (board_grids()). Single-view boards are a degenerate case with one auto-named "Page" view.

## Usage

```
new_dock_board(
  blocks = list(),
  links = list(),
  stacks = list(),
  ...,
  extensions = new_dock_extensions(),
  views = NULL,
  grids = NULL,
  active = NULL,
  options = dock_board_options(),
  ctor = NULL,
```

```
  pkg = NULL,
  class = character()
)

is_dock_board(x)

as_dock_board(x, ...)

dock_extensions(x)

dock_extensions(x) <- value

dock_ext_ids(x)

extension_ids(x, class = NULL)

dock_board_options()
```

## Arguments

| | |
|---|---|
| blocks | Set of blocks |
| links | Set of links |
| stacks | Set of stacks |
| ... | Further (metadata) attributes |
| extensions | Dock extensions. Each is keyed by its list name, or by its class stripped of the _extension suffix when unnamed (so new_dag_extension() becomes dag); that key names the extension's panel and is what ext() resolves against. |
| views | Per-view membership: a named list keyed by view id (minted when absent), each value a dock_view(), or member panels named with blk() / ext() references or bare ids (as a vector or list). NULL yields a single default view over the board's blocks and extensions. |
| grids | Per-view geometry: a named list keyed by view id whose values are dock_grid()s, or a dock_grids. Optional – a view with no grid entry falls back to a default grid over its members. |
| active | Id of the initially active view (a key of views). Defaults to the first view. Which view is active is a property of the collection, not of an individual view. A served board also honours a ?view=<id> URL query parameter naming a view id, which opens the board on that view and overrides this default for that request. |
| options | Board-level user settings |
| ctor, pkg | Constructor information (used for serialization) |
| class | Optional extension subclass; when supplied, only ids of extensions inheriting from it are returned. |
| x | Board object |
| value | Replacement value |

## Details

Multi-view boards pass `views` (and optionally `grids`); see dock_view() and dock_grid() for the input forms and board_views() for how they combine. Bare block / extension ids are resolved against the board's blocks and extensions. With `views = NULL` the board gets a single default view.

## Value

The constructor `new_dock_board()` returns a `board` object, as does the coercion function `as_dock_board()`. Inheritance can be checked using `is_dock_board()`, which returns a boolean. The `dock_extensions()` and `dock_extensions<-()` accessors return / set the board's `dock_extension` objects. A character vector of IDs is returned by `dock_ext_ids()` and `dock_board_options()` returns a `board_options` object: the board name, a thematic switch, off unless the `thematic` blockr option says otherwise, so that plots can take the board's colours, a light/dark switch, light unless the `dark_mode` blockr option says otherwise, the "Compact" header switch (`new_compact_option()`), off unless the `compact` blockr option says otherwise, the "Views as tabs" switch (`new_view_tabs_option()`), off unless the `view_tabs` blockr option says otherwise, and a switch that opens and closes the rails of every view together (`new_sync_rails_option()`), on unless the `sync_rails` blockr option says otherwise.

## Examples

```
brd <- new_dock_board(c(a = blockr.core::new_dataset_block()))
view_members(board_views(brd)[[1L]])
```

---

# new_dock_extension *Dock extensions*

---

## Description

Functionality of a `dock_board` can be extended by supplying one or more `dock_extension` objects, which essentially provide UI shown in a dock panel that allows for manipulating the board state. A set of dock extensions can be combined into a `dock_extensions` object.

## Usage

```
new_dock_extension(
  server,
  ui,
  name,
  class,
  description = NULL,
  ctor = sys.parent(),
  pkg = NULL,
  options = new_board_options(),
  external_ctrl = FALSE,
  ...
```

```
)

is_dock_extension(x)

validate_extension(x, ...)

extension_ui(x, key, id, ...)

extension_server(x, key, ...)

extension_id(x)

extension_name(x)

extension_ctor(x)

new_dock_extensions(x = list())

is_dock_extensions(x)

validate_extensions(x)

as_dock_extensions(x, ...)

## S3 method for class 'dock_extensions'
as_dock_extensions(x, ...)

## S3 method for class 'dock_extension'
as_dock_extensions(x, ...)

## S3 method for class 'list'
as_dock_extensions(x, ...)

extension_block_callback(x, ...)
```

## Arguments

| | |
|---|---|
| server | A function returning shiny::moduleServer(). Beyond id, it is called with the board handles board, update, view_data and actions, plus extensions – an environment exposing every extension's server result keyed by ID (each carrying its state), so one extension can read another's state via extensions[[id]]. view_data is a reactive holding the live all-views layout (the committed board is board); it is NULL until every view has reported its layout once, so req() it. |
| ui | A function with a single argument (ns) returning a shiny.tag |
| name | Name for extension |
| class | Extension subclass |
| description | Optional extension metadata, surfaced as consumer-neutral documentation (e.g. to the AI assistant): either a free-text string or a structured new_ext_meta() |

| | |
|---|---|
| | object carrying per-variable documentation for a client driving the extension through `modify_extension` |
| `ctor` | Constructor function name |
| `pkg` | Package to look up `ctor` |
| `options` | Board options supplied by an extension |
| `external_ctrl` | Set up external control (experimental). `FALSE` (the default) opts out; `TRUE` exposes every constructor input as externally controllable; a character vector names a subset of them. |
| `...` | Further attributes |
| `x` | Extension object |
| `key` | Container-assigned extension id (list key) |
| `id` | Namespace ID |

## Value

The constructors `new_dock_extension()` and `new_dock_extension()`, as do the coercion function `as_dock_extension()` and `as_dock_extension()`, return objects that inherit from `dock_extension` and `dock_extensions` respectively. This inheritance structure can be checked using `is_dock_extension()` and `is_dock_extensions()`, which both return a boolean. A `dock_extension` can be validated using `validate_extension()` and a `dock_extensions` object using `validate_extensions()`, which return the input object invisibly and throw errors as side-effects. Several getter functions return extension attributes, including `extension_ui()` (a function), `extension_server()` (a function), `extension_id()` (a string), `extension_name()` (a string) and `extension_ctor()` (an object that inherits from `blockr_ctor`). The extension's metadata is carried as an `ext_meta()` and read per component with `ext_desc()`, `ext_args()`, `ext_examples()` and `ext_guidance()`.

## Examples

```
ext <- new_edit_board_extension()
is_dock_extension(ext)
```

---

# new_dock_stack *Colored stacks*

---

## Description

While stacks created via `blockr.core::new_stack()` do not keep track of a color attribute, a `dock_stack` object does. Such objects can be created via `new_dock_stack()`. The color attribute can be extracted using `stack_color()` and set with `stack_color<-()`. A new color suggestion, based on existing colors, is available through `suggest_new_colors()`.

## Usage

```
new_dock_stack(..., color = suggest_new_colors())

is_dock_stack(x)

stack_color(x)

suggest_new_colors(colors = character(), n = 1)

stack_color(x) <- value

as_dock_stack(x, ...)
```

## Arguments

| | |
|---|---|
| `...` | Passed to `blockr.core::new_stack()` |
| `color` | String-valued color value (using hex encoding) |
| `x` | object |
| `colors` | Currently used color values |
| `n` | Number of new colors to generate |
| `value` | Replacement value |

## Value

The constructor `new_dock_stack()` returns a "dock_stack" object, which is a stack object as returned by `blockr.core::new_stack()`, with an additional color attribute. Inheritance can be checked using `is_dock_stack()`, which returns a scalar logical and the color attribute can be set and retrieved using `stack_color<-()` (returns the modified stack object invisibly) and `stack_color()` (returns a string), respectively. Stack objects may be coerced to "dock_stack" using `as_dock_stack()` and finally, a utility function `suggest_new_colors()` which returns a character vector of new colors, based on an existing palette.

---

# new_edit_board_extension

*Edit board extension*

---

## Description

A simplistic example of an extension which can be used for manipulating the board via a table-based UI. Mainly relevant for testing purposes.

## Usage

```
new_edit_board_extension(...)
```

### Arguments

| | |
|---|---|
| `...` | Forwarded to `new_dock_extension()` |

### Value

A board extension object that additionally inherits from `edit_board_extension`.

### Examples

```
ext <- new_edit_board_extension()
is_dock_extension(ext)
```

---

## `new_sync_rails_option` *Rails in step across views*

---

### Description

A board option that opens and closes the rails of every view together. Collapsing or expanding the rail on one edge of a view does the same to the rail on that edge of every other view, and a view visited for the first time opens with its rails the way the views already visited show theirs. Only a rail holding panels takes part: an empty one is not shown. On by default; switched off, each view keeps its rails as they were left there.

### Usage

```
new_sync_rails_option(
  value = blockr_option("sync_rails", TRUE),
  category = "Board options",
  ...
)
```

### Arguments

| | |
|---|---|
| `value` | Logical, whether the rails are kept in step. Defaults to the `sync_rails` blockr option, else `TRUE`. |
| `category` | Options sidebar category. |
| `...` | Passed to `blockr.core::new_board_option()`. |

### Value

A `board_option` object.

### Examples

```
new_sync_rails_option(FALSE)
```

## new_view_tabs_option *Views as tabs*

### Description

A board option that shows the board's views as a line of tabs under the navbar, one tab per view, with the view menu reduced to its chevron at the end of the line. Off by default, it sits with the light/dark switch and "Compact" under "Theme options", and "Show views as tabs" at the foot of the view menu sets it as well. A board saved with the option on shows the tabs from its first paint.

### Usage

```
new_view_tabs_option(
  value = blockr_option("view_tabs", FALSE),
  category = "Theme options",
  label = "Views as tabs",
  ...
)
```

### Arguments

| | |
|---|---|
| value | Logical, whether the views show as tabs. Defaults to the view_tabs blockr option, else FALSE. |
| category | Options sidebar category. |
| label | The option's name, which its switch shows. |
| ... | Passed to blockr.core::new_board_option(). |

### Value

A board_option object.

### Examples

```
new_view_tabs_option(TRUE)
```

## panel_obj_ids *Panel IDs of a grid or layout*

### Description

layout_panel_ids() returns the canonical panel IDs (block_panel-... / ext_panel-...) a dock_grid or dock_layout references; panel_obj_ids() strips those prefixes back to the bare block / extension IDs.

## Usage

```
panel_obj_ids(ids)

layout_panel_ids(layout)
```

## Arguments

| | |
|---|---|
| ids | Character vector of panel IDs. |
| layout | A dock_grid or dock_layout. |

## Value

Character vectors of IDs.

# Index


action_id *(new_action)*, 17
action_triggers *(new_action)*, 17
active_view *(is_dock_grids)*, 13
active_view<- *(is_dock_grids)*, 13
as.character.panel_ref *(blk)*, 2
as_block_handle_id *(dock_id)*, 9
as_block_panel_id *(dock_id)*, 9
as_dock_board *(new_dock_board)*, 20
as_dock_extensions *(new_dock_extension)*, 22
as_dock_grid *(dock-grid)*, 6
as_dock_grid(), *7*
as_dock_handle_id *(dock_id)*, 9
as_dock_layout *(dock-layout)*, 6
as_dock_panel_id *(dock_id)*, 9
as_dock_stack *(new_dock_stack)*, 24
as_dock_view *(is_dock_grids)*, 13
as_dock_view(), *7*
as_ext_handle_id *(dock_id)*, 9
as_ext_panel_id *(dock_id)*, 9
as_obj_id *(dock_id)*, 9
as_panel_ref *(blk)*, 2
as_view_handle_id *(dock_id)*, 9

blk, 2
blk_color *(blks_metadata)*, 4
blk_icon_data_uri *(blks_metadata)*, 4
blks_metadata, 4
block_input_select *(new_action)*, 17
block_status_badge *(blks_metadata)*, 4
blockr.core::block_name(), *14*
blockr.core::new_board(), *20*
blockr.core::new_board_option(), *20*, *26*, *27*
blockr.core::new_stack(), *24*, *25*
blockr.core::serve(), *15*
blockr.ui::block_mark_svg(), *5*
blockr.ui::category_color(), *5*
blockr_app_navbar *(navbar_item)*, 15
board_actions *(new_action)*, 17
board_block_select *(new_action)*, 17
board_grids *(is_dock_grids)*, 13
board_grids(), *20*
board_grids<- *(is_dock_grids)*, 13
board_update, *2*
board_views *(is_dock_grids)*, 13
board_views(), *20*, *22*
board_views<- *(is_dock_grids)*, 13

custom_navbar *(navbar_item)*, 15

default_layout *(dock_grid)*, 7
default_navbar_items *(navbar_item)*, 15
dock-grid, 6
dock-layout, 6
dock_board_options *(new_dock_board)*, 20
dock_ext_ids *(new_dock_board)*, 20
dock_extensions *(new_dock_board)*, 20
dock_extensions<- *(new_dock_board)*, 20
dock_grid, *6*, 7, *7*, *9*, *27*, *28*
dock_grid(), *6*, *14*, *21*, *22*
dock_id, 9
dock_layout, *6*, *9*, *15*, *27*, *28*
dock_view *(is_dock_grids)*, 13
dock_view(), *6*, *7*, *14*, *21*, *22*

ext *(blk)*, 2
ext-meta, 11
ext_args *(ext-meta)*, 11
ext_desc *(ext-meta)*, 11
ext_examples *(ext-meta)*, 11
ext_guidance *(ext-meta)*, 11
ext_meta *(ext-meta)*, 11
ext_meta(), *24*
extension_block_callback *(new_dock_extension)*, 22
extension_ctor *(new_dock_extension)*, 22
extension_description *(ext-meta)*, 11
extension_id *(new_dock_extension)*, 22
extension_ids *(new_dock_board)*, 20

extension_name *(new_dock_extension)*, 22
extension_server *(new_dock_extension)*, 22
extension_ui *(new_dock_extension)*, 22
external_ctrl_vars(), *11*

group *(dock_grid)*, 7
group(), *8*

is_action *(new_action)*, 17
is_action_generator *(new_action)*, 17
is_block_panel_id *(dock_id)*, 9
is_dock_board *(new_dock_board)*, 20
is_dock_extension *(new_dock_extension)*, 22
is_dock_extensions *(new_dock_extension)*, 22
is_dock_grid *(dock-grid)*, 6
is_dock_grids, 13
is_dock_layout *(dock-layout)*, 6
is_dock_panel_id *(dock_id)*, 9
is_dock_stack *(new_dock_stack)*, 24
is_dock_view *(is_dock_grids)*, 13
is_dock_views *(is_dock_grids)*, 13
is_ext_meta *(ext-meta)*, 11
is_ext_panel_id *(dock_id)*, 9
is_navbar_item *(navbar_item)*, 15
is_navbar_items *(navbar_item)*, 15
is_panel_ref *(blk)*, 2

layout_panel_ids *(panel_obj_ids)*, 27

navbar_item, 15
new_action, 17
new_arg_specs(), *12*
new_compact_option, 19
new_compact_option(), *22*
new_dock_board, 20
new_dock_extension, 22
new_dock_extension(), *11*
new_dock_extensions *(new_dock_extension)*, 22
new_dock_layout *(dock-layout)*, 6
new_dock_stack, 24
new_edit_board_extension, 25
new_ext_meta *(ext-meta)*, 11
new_ext_meta(), *23*
new_sync_rails_option, 26
new_sync_rails_option(), *22*
new_view_tabs_option, 27
new_view_tabs_option(), *16*, *22*

panel-ids *(panel_obj_ids)*, 27
panel_obj_ids, 27
panels *(dock_grid)*, 7
panels(), *8*
plugin_navbar_item *(navbar_item)*, 15

rail *(dock_grid)*, 7
rail(), *4*

shiny::moduleServer(), *16*, *18*, *23*
shiny::observeEvent(), *19*
shiny::reactiveVal(), *18*
shiny::selectizeInput(), *18, 19*
shiny::textOutput(), *16*
shiny::uiOutput(), *16*
shiny::updateSelectizeInput(), *19*
sidebar_owned_by *(new_action)*, 17
stack_color *(new_dock_stack)*, 24
stack_color<- *(new_dock_stack)*, 24
suggest_new_colors *(new_dock_stack)*, 24

validate_dock_grid *(dock-grid)*, 6
validate_dock_grids *(is_dock_grids)*, 13
validate_dock_layout *(dock-layout)*, 6
validate_dock_view *(is_dock_grids)*, 13
validate_dock_views *(is_dock_grids)*, 13
validate_extension *(new_dock_extension)*, 22
validate_extensions *(new_dock_extension)*, 22
view_members *(is_dock_grids)*, 13
view_name *(is_dock_grids)*, 13
view_name<- *(is_dock_grids)*, 13
view_names *(is_dock_grids)*, 13