Package: blockr.dock (via r-universe)

June 5, 2026

Title A Docking Layout Manager for 'blockr'

Version 0.1.2

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)

RoxygenNote 7.3.3

Imports blockr.core (>= 0.1.3), blockr.ui, bsicons, shiny, shinyjs, bslib, glue, dockViewR (>= 0.2.1), htmltools, cli, shinyWidgets, jsonlite

Remotes BristolMyersSquibb/blockr.ui, BristolMyersSquibb/blockr.core

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

Config/testthat/edition 3

Collate 'action-class.R' 'action-block.R' 'action-link.R' 'action-sidebar.R' 'action-stack.R' 'action-ui.R' 'action-utils.R' 'block-meta.R' 'block-ui.R' 'board-plugins.R' 'board-server.R' 'board-ui.R' 'dock-board.R' 'dock-stack.R' 'ext-class.R' 'ext-delta.R' 'ext-edit.R' 'ext-ui.R' 'layout-class.R' 'plugin-block.R' 'plugin-serdes.R' 'utils-dock.R' 'utils-id.R' 'utils-misc.R' 'utils-pkg.R' 'utils-serdes.R' 'utils-serve.R' 'utils-ui.R' 'layouts-class.R' 'view-ui.R' 'views-delta.R'

VignetteBuilder knitr

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

Repository https://bristolmyerssquibb.r-universe.dev

Date/Publication 2026-06-05 09:23:48 UTC

RemoteUrl https://github.com/bristolmyerssquibb/blockr.dock

RemoteRef HEAD

RemoteSha 3b6ba86a0ed9da1b4e92cc2db2978680888a3df0

Contents

`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"))
```

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

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.
- `blk_color()`: Produces colors using the Okabe-Ito colorblind-friendly palette for a character vector of block categories.
- `blk_icon_data_uri()`: Processes block icons to add color and turn them into square-shaped icons.

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.

Examples

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

col <- blk_color(meta$category)
blk_icon_data_uri(meta$icon, col)
```

---

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)

as_dock_panel_id(x)

as_obj_id(x)
```

```
as_block_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().

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)))
```

dock_layout *Dock layout*

Description

A dock_layout is the panel arrangement for a single view: a tree of block / extension IDs, with at most one leaf marked as initially active. A board holds a dock_layouts collection (one layout per view); panel content is derived on demand from the board's blocks and extensions, so only the arrangement is stored in a dock_layout. See is_dock_layouts() for the collection-level helpers.

Usage

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

panels(..., active = NULL)

group(..., sizes = NULL)

is_dock_layout(x)

as_dock_layout(x, ...)

## S3 method for class 'dock_layout'
format(x, ..., bare = TRUE)

## S3 method for class 'dock_layout'
print(x, ...)

default_layout(blocks, extensions)

validate_dock_layout(x, blocks = character())
```

Arguments

...	For dock_layout() and group(), layout children (bare IDs, character vectors, lists, panels(), or group()). For panels(), panel IDs. 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).

`name` For `dock_layout()`, an optional display label for the view (free-form). When omitted, a label is derived from the view's id. The view's id is the list name in `new_dock_board(layouts = list(...))`, minted when absent and unique across the views of a `dock_layouts`.

`active` For `panels()`, the ID of the tab to open by default.

`x` Object

`bare` For `format()` / `print()`, drop the `block_panel-` / `ext_panel-` prefixes from panel IDs (see `panel_obj_ids()`).

`blocks, extensions`
Dock board components. For `default_layout()` the components to arrange; for `as_dock_layout()`, optional, used to resolve bare IDs and validate the result.

Details

Construct a layout with:

- `dock_layout(...)`: the page-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.
- `default_layout(blocks, extensions)` produces the default two-row arrangement (extensions on top, blocks below) for a board.

`dock_layout()` accepts `orientation = "horizontal" | "vertical"` for the top-level split direction, `sizes` for the root-branch ratios, and `name` for the view's display label. In `new_dock_board(layouts = list(...))` the list name is the view's stable *id* (the container's key, like a block id), minted when absent; `name` sets the free-form display label on the view itself. When no name is given, one is derived from the id for display. Which view starts active is a property of the collection, not of any one layout: pass `new_dock_board(active = )` (a view id) to choose it, defaulting to the first.

A *view* is the conceptual page-level container; a *layout* is the panel arrangement inside a view. The dockview-shape `grid` + `panels` payload that dockViewR consumes is an internal projection of a `dock_layout` against the board's blocks and extensions; it is not a public type.

`as_dock_layout()` coerces to a `dock_layout`: a `dock_layout` (identity), a `board` (its active layout), or a spec list (`as.list()` of a layout, or a parsed `layout_to_json()` string). Pass `blocks` / `extensions` to resolve bare IDs to canonical panel IDs and validate. `as.list()` of a `dock_layout` returns that spec list. The JSON-string boundary is `layout_to_json()` / `layout_from_json()`.

Value

`dock_layout()` and `default_layout()` return a `dock_layout` object. `panels()` returns a `dock_panels` node and `group()` returns a `dock_group` node — both are layout sub-trees usable inside `dock_layout()`

/ `group()`. `as_dock_layout()` returns a `dock_layout` (from a board or a spec list); `as.list()` of a `dock_layout` returns the spec list. `is_dock_layout()` returns a boolean. `validate_dock_layout()` returns its input and throws on error.

Examples

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

exts <- list(
  edit = new_edit_board_extension()
)

# The default arrangement for a given set of blocks and extensions
default_layout(blks, exts)

# Tabbed leaf with an explicit open tab
panels("a", "b", "edit_board_extension", active = "edit_board_extension")

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

# Composing them inside a layout
dock_layout(
  "a",
  panels("b", "edit_board_extension", active = "edit_board_extension"),
  sizes = c(0.3, 0.7)
)

# Vertical top-level split
dock_layout("a", "b", orientation = "vertical")
```

---

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"),
  ...
)

block_registry_selectize(id, blocks = list_blocks())

board_select(id, blocks, selected = NULL, ...)
```

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
blocks	Character vector of block registry IDs
selected	Character vector of pre-selected block (registry) IDs

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).

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 utilities `block_input_select()`, `block_registry_selectize()` and `board_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.

`block_registry_selectize()`

This creates UI for a block registry selector via `shiny::selectizeInput()` and returns an object that inherits from `shiny.tag`.

`board_select()`

Block selection UI, enumerating all blocks in a board is available as `board_select()`. An object that inherits from `shiny.tag` is returned, which contains the result from a `shiny::selectizeInput()` call.

`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 panel arrangement (as `layouts`). The `layouts` field is always stored internally as a `dock_layouts` collection (multi-view); single-page 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(),
  layouts = default_layout(blocks, extensions),
  active = NULL,
  options = dock_board_options(),
  ctor = NULL,
  pkg = NULL,
  class = character()
)

is_dock_board(x)
```

```
as_dock_board(x, ...)

active_layout(x)

active_layout(x) <- value

dock_extensions(x)

dock_extensions(x) <- value

dock_ext_ids(x)

dock_board_options()
```

Arguments

blocks	Set of blocks
links	Set of links
stacks	Set of stacks
...	Further (metadata) attributes
extensions	Dock extensions
layouts	A named list of per-view arrangements (multi-view), a dock_layout / raw list (single-page), or an existing dock_layouts collection. All forms are normalised to dock_layouts.
active	Id of the initially active view (a key of layouts). Defaults to the first view. Which view is active is a property of the collection, not of an individual layout.
options	Board-level user settings
ctor, pkg	Constructor information (used for serialization)
class	Board sub-class
x	Board object
value	Replacement value

Details

For multi-view boards, pass a named list to layouts = — each name becomes a view, each value is the panel arrangement (a dock_layout() or a raw list of block / extension IDs). For a single-page board, pass a dock_layout or raw list directly. Either way the input is normalised to a dock_layouts, with leaf IDs resolved against the board's blocks and extensions.

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. board_layouts() returns the board's dock_layouts; active_layout() returns the active view's dock_layout and active_layout<-() writes into the active view. 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.

Examples

```
brd <- new_dock_board(c(a = blockr.core::new_dataset_block()))
str(active_layout(brd), max.level = 2)
```

---

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, id, ...)

extension_server(x, ...)

extension_id(x)

extension_name(x)

extension_description(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, dock 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]].
ui	A function with a single argument (ns) returning a shiny.tag
name	Name for extension
class	Extension subclass
description	Optional free-text description of the extension, surfaced as consumer-neutral metadata (e.g. to the AI assistant)
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
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), extension_description() (a string or NULL) and extension_ctor() (an object that inherits from blockr_ctor).

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)
```

panel_obj_ids *Layout serialization and inspection*

Description

Read and write the JSON form of a dock_layout, and inspect the panel IDs it references. These are the canonical accessors for the serialized layout format — downstream tooling should call them rather than re-implement the format.

Usage

```
panel_obj_ids(ids)

layout_panel_ids(layout)

layout_to_json(x, ...)

layout_from_json(x, blocks = NULL, extensions = NULL)
```

Arguments

ids	Character vector of panel IDs.
layout	A dock_layout object.
x	A dock_layout (for layout_to_json()), or a JSON string / parsed spec list (for layout_from_json()).
...	Forwarded to jsonlite::toJSON().
blocks, extensions	Optional board components used to resolve and validate bare IDs in layout_from_json().

Details

layout_to_json() renders a layout as a JSON string; layout_from_json() is the inverse. The shape is a recursive tree: the top object carries orientation, children, an optional sizes, and an optional focus (the panel with current focus); a child is either a bare string (single-panel leaf), an object with panels / optional active (tabbed leaf), or an object with children / optional sizes (nested branch). Sizes are ratios summing to 1, omitted when even.

layout_from_json() accepts a JSON string or an already-parsed spec list and delegates to as_dock_layout(); when blocks / extensions are supplied, bare IDs are resolved to canonical panel IDs and the result is validated (an unknown panel or malformed arrangement throws the usual classed error).

layout_panel_ids() returns the canonical panel IDs (block_panel-. . . / ext_panel-. . .) referenced by a layout; panel_obj_ids() strips those prefixes back to bare block / extension IDs.

Value

layout_to_json() returns a JSON string; layout_from_json() a dock_layout. layout_panel_ids() and panel_obj_ids() return character vectors.

Examples

```
ly <- dock_layout("a", panels("b", "c", active = "c"), sizes = c(0.3, 0.7))

json <- layout_to_json(ly)
cat(json)

identical(layout_from_json(json), ly)
```

---

show_panel *UI utilities*

Description

Exported utilities for manipulating dock panels (i.e. displaying panels).

Usage

```
show_panel(id, board, dock, type = c("block", "extension"))
```

Arguments

id	Object ID
board	Board object
dock	Object available as dock in extensions
type	Either "block" or "extensions", depending on what kind of panel should be shown

Value

NULL, invisibly

Index