Package 'blockr.core'

May 20, 2025

Title Graphical Web-Framework for Data Manipulation and Visualization

Version 0.1.0

Description A framework for data manipulation and visualization using a web-based point and click user interface where analysis pipelines are decomposed into re-usable and parameterizable blocks.

URL https://bristolmyerssquibb.github.io/blockr.core/

BugReports https://github.com/BristolMyersSquibb/blockr.core/issues

License GPL (>= 3)

Encoding UTF-8

RoxygenNote 7.3.2

Imports shiny (>= 1.5.0), DT, bslib, bsicons, utils, jsonlite, vctrs, generics, rlang, htmltools, shinyFiles

Suggests testthat (>= 3.0.0), memuse, withr, grDevices, shinytest2, roxy.shinylive, knitr, rmarkdown, quarto, scoutbaR

Config/testthat/edition 3

VignetteBuilder quarto

NeedsCompilation no

Author Nicolas Bennett [aut, cre],
David Granjon [aut],
Christoph Sax [aut],
Karma Tarap [ctb],
John Coene [ctb],
Bristol Myers Squibb [fnd]

Maintainer Nicolas Bennett <nicolas@cynkra.com>

Repository CRAN

Date/Publication 2025-05-20 08:10:02 UTC

Contents

`blockr_option` *Blockr Options*

Description

Retrieves options via `base::getOption()` or `base::Sys.getenv()`, in that order, and prefixes the option name passed as `name` with `blockr.` or `blockr_` respectively. Additionally, the name is converted to lower case for `getOption()` and upper case for environment variables. In case no value is available for a given `name`, `default` is returned.

Usage

```
blockr_option(name, default)
```

Arguments

name	Option name
default	Default value

Value

The value set as option `name` or `default` if not set. In case of the option being available only as environment variable, the value will be a string and if available as `base::options()` entry it may be of any R type.

Examples

```
blockr_option("test-example", "default")

options(`blockr.test-example` = "non-default")
blockr_option("test-example", "default")

Sys.setenv(`BLOCKR_TEST-EXAMPLE` = "another value")
tryCatch(
  blockr_option("test-example", "default"),
  error = function(e) conditionMessage(e)
)
options(`blockr.test-example` = NULL)
blockr_option("test-example", "default")

Sys.unsetenv("BLOCKR_TEST-EXAMPLE")
blockr_option("test-example", "default")
```

---

blockr_ser *Serialization utilities*

Description

Object serialization is available via `to_json()`, while de-serialization is available as `from_json()`. Blocks are serialized by writing out information on the constructor used to create the object, combining this with block state information, which constitutes values such that when passed to the constructor the original object can be re-created.

Usage

```
blockr_ser(x, ...)

## S3 method for class 'block'
blockr_ser(x, state = NULL, ...)

## S3 method for class 'blocks'
blockr_ser(x, blocks = NULL, ...)

## S3 method for class 'board_options'
blockr_ser(x, options = NULL, ...)

## S3 method for class 'board'
blockr_ser(x, blocks = NULL, options = NULL, ...)

## S3 method for class 'link'
blockr_ser(x, ...)

## S3 method for class 'links'
blockr_ser(x, ...)

## S3 method for class 'stack'
blockr_ser(x, ...)

## S3 method for class 'stacks'
blockr_ser(x, ...)

blockr_deser(x, ...)

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

## S3 method for class 'block'
blockr_deser(x, data, ...)

## S3 method for class 'blocks'
blockr_deser(x, data, ...)

## S3 method for class 'board'
blockr_deser(x, data, ...)

## S3 method for class 'link'
blockr_deser(x, data, ...)

## S3 method for class 'links'
blockr_deser(x, data, ...)

## S3 method for class 'stack'
```

```
blockr_deser(x, data, ...)

## S3 method for class 'stacks'
blockr_deser(x, data, ...)

## S3 method for class 'board_options'
blockr_deser(x, data, ...)

to_json(x, ...)

from_json(x)
```

Arguments

x	Object to (de)serialize
...	Generic consistency
state	Object state (as returned from an expr_server)
blocks	Block states (NULL defaults to values from ctor scope)
options	Board option values (NULL means default values)
data	List valued data (converted from JSON)

Details

Helper functions blockr_ser() and blockr_deser() are implemented as generics and perform most of the heavy lifting for (de-)serialization: representing objects as easy-to-serialize (nested) lists containing mostly strings and no objects which are hard/impossible to truthfully re-create in new sessions (such as environments).

Value

Serialization helper function blockr_ser() returns lists, which for most objects contain slots object and payload, where object contains a class vector which is used by blockr_deser() to instantiate an empty object of that class and use S3 dispatch to identify the correct method that, given the content in payload, can re-create the original object. These are wrapped by to_json(), which returns JSON and from_json() which can consume JSON and returns the original object.

Examples

```
blk <- new_dataset_block("iris")

blockr_ser(blk)
to_json(blk)

all.equal(blk, blockr_deser(blockr_ser(blk)), check.environment = FALSE)
all.equal(blk, from_json(to_json(blk)), check.environment = FALSE)
```

`block_name` *Block utilities*

Description

Several utilities for working (and manipulating) `block` objects are exported and developers are encouraged to use these instead of relying on object implementation to extract or modify attributes. If functionality for working with blocks in lacking, please consider opening an issue.

Usage

```
block_name(x)

block_name(x) <- value

validate_data_inputs(x, data)

block_inputs(x)

block_arity(x)
```

Arguments

`x`	An object inheriting from `"block"`
`value`	New value
`data`	Data input values

Value

Return types vary among the set of exported utilities:

- `block_name()`: string valued block name,
- `block_name<-()`: `x` (invisibly),
- `validate_data_inputs()`: `NULL` if no validator is set and the result of the validator function otherwise,
- `block_inputs()`: a (possibly empty) character vector of data input names,
- `block_arity()`: a scalar integer with `NA` in case of variadic behavior.

Block name

Each block can have a name (by default constructed from the class vector) intended for users to easily identify different blocks. This name can freely be changed during the lifetime of a block and no uniqueness restrictions are in place. The current block name can be retrieved with `block_name()` and set as `block_name(x) <- "some name"`.

Input validation

Data input validation is available via `validate_data_inputs()` which uses the (optional) validator function passed to `new_block()` at construction time. This mechanism can be used to prevent premature evaluation of the block expression as this might lead to unexpected errors.

Block arity/inputs

The set of explicit (named) data inputs for a block is available as `block_inputs()`, while the block arity can be queried with `block_arity()`. In case of variadic blocks (i.e. blocks that take a variable number of inputs like for example a block providing `base::rbind()`-like functionality), `block_arity()` returns `NA` and the special block server function argument `...args`, signalling variadic behavior is stripped from `block_inputs()`.

Examples

```
blk <- new_dataset_block()
block_name(blk)
block_name(blk) <- "My dataset block"
block_name(blk)

block_inputs(new_dataset_block())
block_arity(new_dataset_block())

block_inputs(new_merge_block())
block_arity(new_merge_block())

block_inputs(new_rbind_block())
block_arity(new_rbind_block())
```

`block_server` *Block server*

Description

A block is represented by several (nested) shiny modules and the top level module is created using the `block_server()` generic. S3 dispatch is offered as a way to add flexibility, but in most cases the default method for the `block` class should suffice at top level. Further entry points for customization are offered by the generics `expr_server()` and `block_eval()`, which are responsible for initializing the block "expression" module (i.e. the block server function passed in `new_block()`) and block evaluation (evaluating the interpolated expression in the context of input data), respectively.

Usage

```
block_server(id, x, data = list(), ...)

## S3 method for class 'block'
block_server(
```

```
  id,
  x,
  data = list(),
  block_id = id,
  edit_block = NULL,
  board = reactiveValues(),
  update = reactiveVal(),
  ...
)

expr_server(x, data, ...)

block_eval(x, expr, data, ...)
```

Arguments

id	Namespace ID
x	Object for which to generate a shiny::moduleServer()
data	Input data (list of reactives)
...	Generic consistency
block_id	Block ID
edit_block	Block edit plugin
board	Reactive values object containing board information
update	Reactive value object to initiate board updates
expr	Quoted expression to evaluate in the context of data

Details

The module returned from block_server(), at least in the default implementation, provides much of the essential but block-type agnostic functionality, including data input validation (if available), instantiation of the block expression server (handling the block-specific functionality, i.e. block user inputs and expression), and instantiation of the edit_block module (if passed from the parent scope).

A block is considered ready for evaluation whenever input data is available that satisfies validation (validate_data_inputs()) and nonempty state values are available (unless otherwise instructed via allow_empty_state in new_block()). Conditions raised during validation and evaluation are caught and returned in order to be surfaced to the app user.

Block-level user inputs (provided by the expression module) are separated from output, the behavior of which can be customized via the block_output() generic. The block_ui() generic can then be used to control rendering of outputs.

Value

Both block_server() and expr_server() return shiny server module (i.e. a call to shiny::moduleServer()), while block_eval() evaluates an interpolated (w.r.t. block "user" inputs) block expression in the context of block data inputs.

block_ui *Block UI*

Description

The UI associated with a block is created via the generics `expr_ui()` and `block_ui()`. The former is mainly responsible for user inputs that are specific to every block type (i.e. a `subset_block` requires different user inputs compared to a `head_block`, see `new_transform_block()`) and essentially calls the UI function passed as `ui` to `new_block()`. UI that represents block outputs typically is shared among similar block types (i.e. blocks with shared inheritance structure, such as `subset_block` and `head_block`, which both inherit from `transform_block`). This type of UI us created by `block_ui()` and block inheritance is used to deduplicate shared functionality (i.e. by implementing a method for the `transform_block` class only instead of separate methods for `subset_block` and `head_block`. Working in tandem with `block_ui()`, the generic `block_output()` generates the output to be displayed by the UI portion defined via `block_ui()`.

Usage

```
block_ui(id, x, ...)

expr_ui(id, x, ...)

block_output(x, result, session)

## S3 method for class 'board'
block_ui(id, x, blocks = NULL, edit_ui = NULL, ...)
```

Arguments

`id`	Namespace ID
`x`	Object for which to generate a UI container
`...`	Generic consistency
`result`	Block result
`session`	Shiny session object
`blocks`	(Additional) blocks (or IDs) for which to generate the UI
`edit_ui`	Block edit plugin

Details

The result of `block_output()`, which is evaluated in the `block_server()` context is assigned to `output$result`. Consequently, when referencing the block result in `block_ui()`, this naming convention has to be followed by referring to this as something like `shiny::NS(id, "result")`.

Value

Both `expr_ui()` and `block_ui()` are expected to return shiny UI (e.g. objects wrapped in a `shiny::tagList()`). For rendering the UI, `block_output()` is required to return the result of a shiny render function. For example, a transform block might show the resulting `data.frame` as an HTML table using the DT package. The corresponding `block_ui()` function would then contain UI created by `DT::dataTableOutput()` and rendering in `block_output()` would then be handled by `DT::renderDT()`.

Board-level block UI

While the contents of block-level UI are created by dispatching `block_ui()` on blocks another dispatch on `board` (see `new_board()`) occurs as well. This can be used to control how blocks are integrated into the board UI. For the default board, this uses `bslib::card()` to represent blocks. For boards that extend the default `board` class, control is available for how blocks are displayed by providing a board-specific `block_ui()` method.

---

board_blocks *Board utils*

Description

A set of utility functions is available for querying and manipulating board components (i.e. blocks, links and stacks). Functions for retrieving and modifying board options are documented in `new_board_options()`.

Usage

```
board_blocks(x)

board_blocks(x) <- value

board_block_ids(x)

rm_blocks(x, rm)

board_links(x)

board_links(x) <- value

board_link_ids(x)

modify_board_links(x, add = NULL, rm = NULL, mod = NULL)

board_stacks(x)

board_stacks(x) <- value

board_stack_ids(x)
```

```
modify_board_stacks(x, add = NULL, rm = NULL, mod = NULL)

available_stack_blocks(
  x,
  stacks = board_stacks(x),
  blocks = board_stack_ids(x)
)
```

Arguments

x	Board
value	Replacement value
rm	Block/link/stack IDs to remove
add	Links/stacks to add
mod	Link/stacks to modify
blocks, stacks	Sets of blocks/stacks

Value

Functions for retrieving, as well as updating components (board_blocks()/board_links()/board_stacks() and board_blocks<-()/board_links<-()/board_stacks<-()) return corresponding objects (i.e. blocks, links and stacks), while ID getters (board_block_ids(), board_link_ids() and board_stack_ids()) return character vectors, as does available_stack_blocks(). Convenience functions rm_blocks(), modify_board_links() and modify_board_stacks() return an updated board object.

Blocks

Board blocks can be retrieved using board_blocks() and updated with the corresponding replacement function board_blocks<-(). If just the current board IDs are of interest, board_block_ids() is available as short for names(board_blocks(x)). In order to remove block(s) from a board, the (generic) convenience function rm_blocks() is exported, which takes care (in the default implementation for board) of also updating links and stacks accordingly. The more basic replacement function board_blocks<-() might fail at validation of the updated board object if an inconsistent state results from an update (e.g. a block referenced by a stack is no longer available).

Links

Board links can be retrieved using board_links() and updated with the corresponding replacement function board_links<-(). If only links IDs are of interest, this is available as board_link_ids(), which is short for names(board_links(x)). A (generic) convenience function for all kinds of updates to board links in one is available as modify_board_links(). With arguments add, rm and mod, links can be added, removed or modified in one go.

Stacks

Board stacks can be retrieved using board_stacks() and updated with the corresponding replacement function board_stacks<-(). If only the stack IDs are of interest, this is available as

board_stack_ids(), which is short for names(board_stacks(x)). A (generic) convenience function to update stacks is available as modify_board_stacks(), which can add, remove and modify stacks depending on arguments passed as add, rm and mod. If block IDs that are not already associated with a stack (i.e. "free" blocks) are of interest, this is available as available_stack_blocks().

Examples

```
brd <- new_board(
  c(
     a = new_dataset_block(),
     b = new_subset_block()
  ),
  list(from = "a", to = "b")
)

board_blocks(brd)
board_block_ids(brd)

board_links(brd)
board_link_ids(brd)

board_stacks(brd)
board_stack_ids(brd)
```

---

board_options *Board options*

Description

User settings at the board level are managed by a board_options object. This can be constructed via new_board_options() and in case the set of user options is to be extended, the constructor is designed with sub-classing in mind. Consequently, the associated validator validate_board_options() is available as S3 generic. Inheritance checking is available as is_board_options() and coercion as as_board_options(). The currently set options for a board object can be retrieved with board_options() and option names are available as list_board_options(), which is short for names(board_options(.)). Finally, in order to extract the value of a specific option, board_option() can be used.

Usage

```
board_options(x)

new_board_options(
  board_name = "Board",
  n_rows = blockr_option("n_rows", 50L),
  page_size = blockr_option("page_size", 5L),
  filter_rows = blockr_option("filter_rows", FALSE),
  dark_mode = blockr_option("dark_mode", NULL),
```

```
  ...,
  class = character()
)

is_board_options(x)

as_board_options(x)

## S3 method for class 'board_options'
as_board_options(x)

## Default S3 method:
as_board_options(x)

validate_board_options(x)

## S3 method for class 'board_options'
validate_board_options(x)

list_board_options(x)

board_option(opt, x)
```

Arguments

x	Board options object
board_name	String valued board name
n_rows, page_size	Number of rows and page size to show for tabular block previews
filter_rows	Enable filtering of rows in tabular block previews
dark_mode	Toggle between dark and light modes
...	Further options
class	Optional sub-class
opt	Board option

Value

All of `new_board_options()`, `as_board_options()` and `board_options()` return a `board_options` object, as does the validator `validate_board_options()`, which is typically called for side effects of throwing errors is validation does not pass. Inheritance checking as `is_board_options()` returns a scalar logical, while `list_board_options()` returns a character vector of option names. Finally, `board_option()` returns the current value for a specific board option, which in principle may be any R object, but typically we have values such as strings or scalar integers and logicals.

Examples

```
opt <- new_board_options()
```

```
is_board_options(opt)
list_board_options(opt)

board_option("page_size", opt)
```

`board_server` *Board server*

Description

A call to `board_server()`, dispatched on objects inheriting from `board`, returns a `shiny::moduleServer()`, containing all necessary logic to manipulate board components via UI. Extensibility over currently available functionality is provided in the form of S3, where a `board_server()` implementation of `board` sub-classes may be provided, as well as via a plugin architecture and callback functions which can be used to register additional observers.

Usage

```
board_server(id, x, ...)

## S3 method for class 'board'
board_server(id, x, plugins = list(), callbacks = list(), ...)
```

Arguments

`id`	Parent namespace
`x`	Board
`...`	Generic consistency
`plugins`	Board plugins as modules
`callbacks`	Single (or list of) callback function(s), called only for their side-effects)

Value

A `board_server()` implementation (such as the default for the `board` base class) is expected to return a `shiny::moduleServer()`.

board_ui.board_options *Board UI*

Description

As counterpart to board_server(), board_ui() is responsible for rendering UI for a board module. This top-level entry point for customizing board appearance and functionality can be overridden by sub-classing the boar object and providing an implementation for this sub-class. Such an implementation is expected to handle UI for plugins and all board components, including blocks, links and stacks, but may rely on functionality that generates UI for these components, such as block_ui() or stack_ui(), as well as already available UI provided by plugins themselves.

Usage

```
## S3 method for class 'board_options'
board_ui(id, x, ...)

## S3 method for class 'board_options'
update_ui(x, session, ...)

board_ui(id, x, ...)

## S3 method for class 'board'
board_ui(id, x, plugins = list(), ...)

## S3 method for class '`NULL`'
board_ui(id, x, ...)

insert_block_ui(id, x, blocks = NULL, ...)

## S3 method for class 'board'
insert_block_ui(id, x, blocks = NULL, ...)

remove_block_ui(id, x, blocks = NULL, ...)

## S3 method for class 'board'
remove_block_ui(id, x, blocks = NULL, ...)

update_ui(x, session, ...)

## S3 method for class 'board'
update_ui(x, session, ...)
```

Arguments

id	Namespace ID

x	Board
...	Generic consistency
session	Shiny session
plugins	UI for board plugins
blocks	(Additional) blocks (or IDs) for which to generate the UI

Details

Dynamic UI updates are handled by functions `insert_block_ui()` and `remove_block_ui()` for adding and removing block-level UI elements to and from `board` UI, whenever blocks are added or removed. The lightly more nondescript updated function `update_ui()` is intended for board-level UI updates, which is currently only needed when restoring from a saved state and board option UI needs to be adjusted accordingly. All these update functions are provided as S3 generics with implementations for `board` and can be extended if so desired.

Value

A `board_ui()` implementation is expected to return shiny::tag or `shiny::tagList()` objects, while updater functions (`insert_block_ui()`, `remove_block_ui()` and `update_ui()`) are called for their side effects (which includes UI updates such as `shiny::insertUI()`, `shiny::removeUI()`) and return the board object passed as `x` invisibly.

---

edit_block *Plugin module for editing board blocks*

Description

Logic and user experience for editing block attributes such as block titles can be customized or enhanced by providing an alternate version of this plugin. The default implementation only handles block titles, but if further (editable) block attributes are to be introduced, corresponding UI and logic can be included here. In addition to blocks titles, this default implementation provides UI for removing, as well as inserting blocks before or after the current one.

Usage

```
edit_block(server = edit_block_server, ui = edit_block_ui)

edit_block_server(id, block_id, board, update, ...)

edit_block_ui(x, id, ...)

block_summary(x, data)

## S3 method for class 'block'
block_summary(x, data)
```

Arguments

server, ui	Server/UI for the plugin module
id	Namespace ID
block_id	Block ID
board	Reactive values object containing board information
update	Reactive value object to initiate board updates
...	Extra arguments passed from parent scope
x	Block
data	Result data

Value

A plugin container inheriting from `edit_block` is returned by `edit_block()`, while the UI component (e.g. `edit_block_ui()`) is expected to return shiny UI (i.e. `shiny::tagList()`) and the server component (i.e. `edit_block_server()`) is expected to return `NULL`.

`edit_stack` *Plugin module for editing board stacks*

Description

Logic and user experience for editing stack attributes such as stack names can be customized or enhanced by providing an alternate version of this plugin. The default implementation only handles stack names, but if further (editable) stack attributes are to be introduced, corresponding UI and logic can be included here. In addition to stack names, this default implementation provides UI for removing the current stack.

Usage

```
edit_stack(server = edit_stack_server, ui = edit_stack_ui)

edit_stack_server(id, stack_id, board, update, ...)

edit_stack_ui(id, x, ...)
```

Arguments

server, ui	Server/UI for the plugin module
id	Namespace ID
stack_id	Stack ID
board	Reactive values object containing board information
update	Reactive value object to initiate board updates
...	Extra arguments passed from parent scope
x	Stack

Value

A plugin container inheriting from `edit_stack` is returned by `edit_stack()`, while the UI component (e.g. `edit_stack_ui()`) is expected to return shiny UI (i.e. `shiny::tagList()`) and the server component (i.e. `edit_stack_server()`) is expected to return `NULL`.

`generate_code` *Code generation plugin module*

Description

All code necessary for reproducing a data analysis as set up in blockr can be made available to the user. Several ways of providing such a script or code snippet are conceivable and currently implemented, we have a modal with copy-to-clipboard functionality. This is readily extensible, for example by offering a download button, by providing this functionality as a `generate_code` module.

Usage

```
generate_code(server = generate_code_server, ui = generate_code_ui)

generate_code_server(id, board, ...)

generate_code_ui(id, board)
```

Arguments

`server, ui`	Server/UI for the plugin module
`id`	Namespace ID
`board`	The initial `board` object
`...`	Extra arguments passed from parent scope

Value

A plugin container inheriting from `generate_code` is returned by `generate_code()`, while the UI component (e.g. `generate_code_ui()`) is expected to return shiny UI (i.e. `shiny::tagList()`) and the server component (i.e. `generate_code_server()`) is expected to return `NULL`.

`is_acyclic.board` *Graph utils*

Description

Block dependencies are represented by DAGs and graph utility functions `topo_sort()` and `is_acyclic()` are used to create a topological ordering (implemented as DFS) of blocks and to check for cycles. An adjacency matrix corresponding to a board is available as `as.matrix()`.

Usage

```
## S3 method for class 'board'
is_acyclic(x)

## S3 method for class 'links'
is_acyclic(x)

topo_sort(x)

is_acyclic(x)

## S3 method for class 'matrix'
is_acyclic(x)
```

Arguments

`x`	Object

Value

Topological ordering via `topo_sort()` returns a character vector with sorted node IDs and the generic function `is_acyclic()` is expected to return a scalar logical value.

Examples

```
brd <- new_board(
  c(
     a = new_dataset_block(),
     b = new_dataset_block(),
     c = new_scatter_block(),
     d = new_subset_block()
  ),
  list(from = c("a", "d"), to = c("d", "c"))
)

as.matrix(brd)
topo_sort(brd)
is_acyclic(brd)
```

`manage_blocks` *Plugin module for managing board blocks*

Description

Logic and user experience for adding/removing blocks to the board can be customized or enhanced by providing an alternate version of this plugin. The default implementation provides a modal-based UI with simple shiny inputs such as drop-downs and text fields.

Usage

```
manage_blocks(server = manage_blocks_server, ui = manage_blocks_ui)

manage_blocks_server(id, board, update, ...)

manage_blocks_ui(id, board)
```

Arguments

`server`, `ui`	Server/UI for the plugin module
`id`	Namespace ID
`board`	The initial `board` object
`update`	Reactive value object to initiate board updates
`...`	Extra arguments passed from parent scope

Details

Updates are mediated via the `shiny::reactiveVal()` object passed as `update`, where block updates are communicated as list entry `blocks` with components `add` and `rm`, where

- `add` may be `NULL` or a `block` object (block IDs may not already exist),
- `rm` may be `NULL` or a string (of existing block IDs).

Value

A plugin container inheriting from `manage_blocks` is returned by `manage_blocks()`, while the UI component (e.g. `manage_blocks_ui()`) is expected to return shiny UI (i.e. `shiny::tagList()`) and the server component (i.e. `manage_blocks_server()`) is expected to return `NULL`.

`manage_links` *Plugin module for managing board links*

Description

Logic and user experience for adding new, removing and modifying existing links to/from the board can be customized or enhanced by providing an alternate version of this plugin. The default implementation provides a table-based UI, presented in a modal.

Usage

```
manage_links(server = manage_links_server, ui = manage_links_ui)

manage_links_server(id, board, update, ...)

manage_links_ui(id, board)
```

Arguments

`server`, `ui`	Server/UI for the plugin module
`id`	Namespace ID
`board`	The initial `board` object
`update`	Reactive value object to initiate board updates
`...`	Extra arguments passed from parent scope

Details

Updates are mediated via the `shiny::reactiveVal()` object passed as `update`, where link updates are communicated as list entry `stacks` with components `add`, `rm` or `mod`, where

- `add` is either `NULL` or a `links` object (link IDs may not already exists),
- `rm` is either `NULL` or a character vector of (existing) link IDs,
- `mod` is either `NULL` or a `links` object (where link IDs must already exist).

Value

A plugin container inheriting from `manage_links` is returned by `manage_links()`, while the UI component (e.g. `manage_links_ui()`) is expected to return shiny UI (i.e. `shiny::tagList()`) and the server component (i.e. `manage_links_server()`) is expected to return `NULL`.

manage_stacks *Plugin module for managing board stacks*

Description

Logic and user experience for adding new, removing and modifying existing stacks to/from the board can be customized or enhanced by providing an alternate version of this plugin. The default implementation provides a table-based UI, presented in a modal.

Usage

```
manage_stacks(server = manage_stacks_server, ui = manage_stacks_ui)

manage_stacks_server(id, board, update, ...)

manage_stacks_ui(id, board)
```

Arguments

server, ui	Server/UI for the plugin module
id	Namespace ID
board	The initial board object
update	Reactive value object to initiate board updates
...	Extra arguments passed from parent scope

Details

Updates are mediated via the shiny::reactiveVal() object passed as update, where stack updates are communicated as list entry stacks with components add, rm or mod, where

- add is either NULL or a stacks object (stack IDs may not already exists),
- rm is either NULL or a character vector of (existing) stack IDs,
- mod is either NULL or a stacks object (where stack IDs must already exist).

Value

A plugin container inheriting from manage_stacks is returned by manage_stacks(), while the UI component (e.g. manage_stacks_ui()) is expected to return shiny UI (i.e. shiny::tagList()) and the server component (i.e. manage_stacks_server()) is expected to return NULL.

new_block *Blocks*

Description

Steps in a data analysis pipeline are represented by blocks. Each block combines data input with user inputs to produce an output. In order to create a block, which is implemented as a shiny module, we require a server function, a function that produces some UI and a class vector.

Usage

```
new_block(
  server,
  ui,
  class,
  ctor,
  ctor_pkg,
  dat_valid = NULL,
  allow_empty_state = FALSE,
  name = NULL,
  ...
)

is_block(x)

as_block(x, ...)

blocks(...)

is_blocks(x)

as_blocks(x, ...)
```

Arguments

server	A function returning shiny::moduleServer()
ui	A function with a single argument (ns) returning a shiny.tag
class	Block subclass
ctor	String-valued constructor name or function/frame number (mostly for internal use or when defining constructors for virtual classes)
ctor_pkg	String-valued package name when passing a string-valued constructor name or NULL
dat_valid	(Optional) input data validator
allow_empty_state	Either TRUE, FALSE or a character vector of state values that may be empty while still moving forward with block eval

name	Block name
...	Further (metadata) attributes
x	An object inheriting from "block"

Details

A block constructor may have arguments, which taken together define the block state. It is good practice to expose all user-selectable arguments of a block (i.e. everything excluding the "data" input) as block arguments such that block can be fully initialized via the constructor. Some default values are required such that blocks can be constructed via constructor calls without arguments. Where it is sensible to do so, specific default values are acceptable, but if in any way data dependent, defaults should map to an "empty" input. For example, a block that provides utils::head() functionality, one such argument could be n and a reasonable default value could be 6L (in line with corresponding default S3 method implementation). On the other hand, a block that performs a base::merge() operation might expose a by argument, but a general purpose default value (that does not depend on the data) is not possible. Therefore, new_merge_block() has by = character().

The return value of a block constructor should be the result of a call to new_block() and ... should be contained in the constructor signature such that general block arguments (e.g. name) are available from the constructor.

Value

Both new_block() and as_block() return an object inheriting from block, while is_block() returns a boolean indicating whether an object inherits from block or not. Block vectors, created using blocks(), as_blocks(), or by combining multiple blocks using base::c() all inherit frm blocks and iss_block() returns a boolean indicating whether an object inherits from blocks or not.

Server

The server function (passed as server) is expected to be a function that returns a shiny::moduleServer(). This function is expected to have at least an argument id (string-valued), which will be used as the module ID. Further arguments may be used in the function signature, one for each "data" input. A block implementing utils::head() for example could have a single extra argument data, while a block that performs base::merge() requires two extra arguments, e.g. x and y. Finally, a variadic block, e.g. a block implementing something like base::rbind(), needs to accommodate for an arbitrary number of inputs. This is achieved by passing a shiny::reactiveValues() object as ...args and thus such a variadic block needs ...args as part of the server function signature. All per-data input arguments are passed as shiny::reactive() or shiny::reactiveVal() objects.

The server function may implement arbitrary shiny logic and is expected to return a list with components expr and state. The expression corresponds to the R code necessary to perform the block task and is expected to be a reactive quoted expression. It should contain user-chosen values for all user inputs and placeholders for all data inputs (using the same names for data inputs as in the server function signature). Such an expression for a base::merge() block could be created using base::bquote() as

```
bquote(
```

```
  merge(x, y, by = .(cols)),
  list(cols = current_val())
}
```

where `current_val()` is a reactive that evaluates to the current user selection of the `by` columns. This should then be wrapped in a `shiny::reactive()` call such that `current_val()` can be evaluated whenever the current expression is required.

The `state` component is expected to be a named list with either reactive or "static" values. In most cases, components of `state` will be reactives, but it might make sense in some scenarios to have constructor arguments that are not exposed via UI components but are fixed at construction time. An example for this could be the `dataset_block` implementation where we have constructor arguments `dataset` and `package`, but only expose `dataset` as UI element. This means that `package` is fixed at construction time. Nevertheless, `package` is required as state component, as this is used for re-creating blocks from saved state.

State component names are required to match block constructor arguments and re-creating saved objects basically calls the block constructor with values obtained from block state.

UI

Block UI is generated using the function passed as `ui` to the `new_block` constructor. This function is required to take a single argument `id` and shiny UI components have to be namespaced such that they are nested within this ID (i.e. by creating IDs as `shiny::NS(id, "some_value")`). Some care has to be taken to properly initialize inputs with constructor values. As a rule of thumb, input elements exposed to the UI should have corresponding block constructor arguments such that blocks can be created with a given initial state.

Block UI should be limited to displaying and arranging user inputs to set block arguments. For outputs, use generics `block_output()` and `block_ui()`.

Sub-classing

In addition to the specific class of a block, the core package uses virtual classes to group together blocks with similar behavior (e.g. `transform_block`) and makes use of this inheritance structure in S3 dispatch for methods like `block_output()` and `block_ui()`. This pattern is not required but encouraged.

Initialization/evaluation

Some control over when a block is considered "ready for evaluation" is available via arguments `dat_valid` and `allow_empty_state`. Data input validation can optionally be performed by passing a predicate function with the same arguments as in the server function (not including `id`) and the block expression will not be evaluated as long as this function throws an error.

Other conditions (messages and warnings) may be thrown as will be caught and displayed to the user but they will not interrupt evaluation. Errors are safe in that they will be caught as well but the will interrupt evaluation as long as block data input does not satisfy validation.

Block vectors

Multiple blocks can be combined into a `blocks` object, a container for an (ordered) set of blocks. Block IDs are handled at the `blocks` level which will ensure uniqueness.

Examples

```
new_identity_block <- function() {
  new_transform_block(
    function(id, data) {
      moduleServer(
        id,
        function(input, output, session) {
          list(
            expr = reactive(quote(identity(data))),
            state = list()
          )
        }
      )
    },
    function(id) {
      tagList()
    },
    class = "identity_block"
  )
}

blk <- new_identity_block()
is_block(blk)

blks <- c(a = new_dataset_block(), b = new_subset_block())

is_block(blks)
is_blocks(blks)

names(blks)

tryCatch(
  names(blks["a"]) <- "b",
  error = function(e) conditionMessage(e)
)
```

---

new_board *Board*

Description

A set of blocks, optionally connected via links and grouped into stacks are organized as a `board` object. Boards are constructed using `new_board()` and inheritance can be tested with `is_board()`, while validation is available as (generic function) `validate_board()`. This central data structure can be extended by adding further attributes and sub-classes. S3 dispatch is used in many places to control how the UI looks and feels and using this extension mechanism, UI aspects can be customized to user requirements. Several utilities are available for retrieving and modifying block attributes (see `board_blocks()`).

Usage

```
new_board(
  blocks = list(),
  links = list(),
  stacks = list(),
  options = new_board_options(),
  ...,
  class = character()
)

validate_board(x)

is_board(x)
```

Arguments

blocks	Set of blocks
links	Set of links
stacks	Set of stacks
options	Board-level user settings
...	Further (metadata) attributes
class	Board sub-class
x	Board object

Value

The board constructor `new_board()` returns a `board` object, as does the validator `validate_board()`, which typically is called for side effects in the form of errors. Inheritance checking as `is_board()` returns a scalar logical.

Examples

```
brd <- new_board(
  c(
     a = new_dataset_block(),
     b = new_subset_block()
  ),
  list(from = "a", to = "b")
)

is_board(brd)
```

`new_data_block` *Data block constructors*

Description

Data blocks typically do not have data inputs and represent root nodes in analysis graphs. Intended as initial steps in a pipeline, such blocks are responsible for providing down-stream blocks with data.

Usage

```
new_data_block(server, ui, class, ctor = sys.parent(), ...)

new_dataset_block(dataset = character(), package = "datasets", ...)

new_static_block(data, ...)
```

Arguments

`server`	A function returning `shiny::moduleServer()`
`ui`	A function with a single argument (`ns`) returning a `shiny.tag`
`class`	Block subclass
`ctor`	String-valued constructor name or function/frame number (mostly for internal use or when defining constructors for virtual classes)
`...`	Forwarded to `new_data_block()` and `new_block()`
`dataset`	Selected dataset
`package`	Name of an R package containing datasets
`data`	Data (used directly as block result)

Value

All blocks constructed via `new_data_block()` inherit from `data_block`.

Dataset block

This data block allows to select a dataset from a package, such as the datasets package available in most R installations as one of the packages with "recommended" priority. The source package can be chosen at time of block instantiation and can be set to any R package, for which then a set of candidate datasets is computed. This includes exported objects that inherit from `data.frame`.

Static block

Mainly useful for testing and examples, this block simply returns the data with which it was initialized. Serialization of static blocks is not allowed and exported code will not be self-contained in the sense that it will not be possible to reproduce results in a script that contains code from a static block.

new_file_block *File block constructors*

Description

Similarly to new_data_block(), blocks created via new_file_block() serve as starting points in analysis pipelines by providing data to down-stream blocks. They typically will not have data inputs and represent root nodes in analysis graphs.

Usage

```
new_file_block(server, ui, class, ctor = sys.parent(), ...)

new_filebrowser_block(
  file_path = character(),
  volumes = c(home = path.expand("~")),
  ...
)

new_upload_block(...)
```

Arguments

server	A function returning shiny::moduleServer()
ui	A function with a single argument (ns) returning a shiny.tag
class	Block subclass
ctor	String-valued constructor name or function/frame number (mostly for internal use or when defining constructors for virtual classes)
...	Forwarded to new_file_block() and new_block()
file_path	File path
volumes	Parent namespace

Value

All blocks constructed via new_file_block() inherit from file_block.

File browser block

In order to make user data available to blockr, this block provides file- upload functionality via shiny::fileInput(). Given that data provided in this way are only available for the life-time of the shiny session, exported code is not self-contained and a script containing code from an upload block is cannot be run in a new session. Also, serialization of upload blocks is currently not allowed as the full data would have to be included during serialization.

Upload block

In order to make user data available to blockr, this block provides file- upload functionality via `shiny::fileInput()`. Given that data provided in this way are only available for the life-time of the shiny session, exported code is not self-contained and a script containing code from an upload block is cannot be run in a new session. Also, serialization of upload blocks is currently not allowed as the full data would have to be included during serialization.

`new_link` *Board links*

Description

Two blocks can be connected via a (directed) link. This means the result from one block is passed as (data) input to the next. Source and destination are identified by `from` and `to` attributes and in case of polyadic receiving blocks, the `input` attribute identified which of the data inputs is the intended destination. In principle, the `link` object may be extended via sub-classing and passing further attributes, but this has not been properly tested so far.

In addition to unique IDs, `links` objects are guaranteed to be consistent in that it is not possible to have multiple links pointing to the same target (combination of `to` and `input` attributes). Furthermore, links behave like edges in a directed acyclic graph (DAG) in that cycles are detected and disallowed.

Usage

```
new_link(from = "", to = "", input = "", ..., class = character())

is_link(x)

as_link(x)

links(...)

is_links(x)

as_links(x)

validate_links(x)
```

Arguments

`from`, `to`	Block ID(s)
`input`	Block argument
`...`	Extensibility
`class`	(Optional) link sub-class
`x`	Links object

Details

A links is created via the `new_link()` constructor and for a vector of links, the container object `links` is provided and a corresponding constructor `links()` exported from the package. Testing whether an object inherits from `link` (or `links`) is available via `is_link()` (or `is_links()`, respectively). Coercion to `link` (and `links`) objects is implemented as `as_link()` (and `as_links()`, respectively). Finally, links can be validated by calling `validate_links()`.

Value

Both `new_link()`/`as_link()`, and `links()`/`as_links()` return `link` and `links` objects, respectively. Testing for inheritance is available as `is_link()`/`is_links()` and validation (for `links`) is performed with `validate_links()`, which returns its input (`x`) or throws an error.

Examples

```
lnks <- links(from = c("a", "b"), to = c("b", "c"), input = c("x", "y"))
is_links(lnks)
names(lnks)

tryCatch(
  c(lnks, new_link("a", "b", "x")),
  error = function(e) conditionMessage(e)
)
tryCatch(
  c(lnks, new_link("b", "a")),
  error = function(e) conditionMessage(e)
)

lnks <- links(a = new_link("a", "b"), b = new_link("b", "c"))
names(lnks)

tryCatch(
  c(lnks, a = new_link("a", "b")),
  error = function(e) conditionMessage(e)
)
```

---

new_parser_block *Parser block constructors*

Description

Operating on results from blocks created via `new_file_block()`, parser blocks read (i.e. "parse") a file and make the contents available to subsequent blocks for further analysis and visualization.

Usage

```
new_parser_block(
  server,
  ui,
  class,
  ctor = sys.parent(),
  dat_valid = is_file,
  ...
)

new_csv_block(sep = ",", quote = "\"", ...)
```

Arguments

server	A function returning shiny::moduleServer()
ui	A function with a single argument (ns) returning a shiny.tag
class	Block subclass
ctor	String-valued constructor name or function/frame number (mostly for internal use or when defining constructors for virtual classes)
dat_valid	(Optional) input data validator
...	Forwarded to new_parser_block() and new_block()
sep, quote	Forwarded to utils::read.table()

Details

If using the default validator for a parser block sub-class (i.e. not overriding the dat_valid argument in the call to new_parser_block()), the data argument corresponding to the input file name must be file in order to match naming conventions in the validator function.

Value

All blocks constructed via new_parser_block() inherit from parser_block.

CSV block

Files in CSV format provided for example by a block created via new_file_block() may be parsed into data.frame by CSV blocks.

`new_plot_block` *Plot block constructors*

Description

Blocks for data visualization using base R graphics can be created via `new_plot_block()`.

Usage

```
new_plot_block(server, ui, class, ctor = sys.parent(), ...)

new_scatter_block(x = character(), y = character(), ...)
```

Arguments

`server`	A function returning `shiny::moduleServer()`
`ui`	A function with a single argument (`ns`) returning a `shiny.tag`
`class`	Block subclass
`ctor`	String-valued constructor name or function/frame number (mostly for internal use or when defining constructors for virtual classes)
`...`	Forwarded to `new_plot_block()` and `new_block()`
`x`, `y`	Columns to place on respective axes

Details

Due to the current block evaluation procedure, where block evaluation is separated from block "rendering" (via `shiny::renderPlot()`) integration of base R graphics requires some mechanism to achieve this decoupling. This is implemented by adding a `plot` attribute to the result of `block_eval()`, generated with `grDevices::recordPlot()` and containing the required information to re-create the plot at a later time. As part of `block_output()`, the attribute is retrieved and passed to `grDevices::replayPlot()`. Consequently, any block that inherits from `plot_block` is required to support this type of decoupling.

Value

All blocks constructed via `new_plot_block()` inherit from `plot_block`.

Scatter block

Mainly for demonstration purposes, this block draws a scattter plot using `base::plot()`. In its current simplistic implementation, apart from axis labels (fixed to the corresponding column names), no further plotting options are available and for any "production" application, a more sophisticated (set of) block(s) for data visualization will most likely be required.

`new_plugin` *Board plugin*

Description

A core mechanism for extending or customizing UX aspects of the board module is a "plugin" architecture. All plugins inherit from `plugin` and a sub-class is assigned to each specific plugin. The "manage blocks" plugin for example has a class vector `c("manage_blocks", "plugin")`. Sets of plugins are handled via a wrapper class `plugins`. Each plugin needs a server component, in most cases accompanied by a UI component and is optionally bundled with a validator function.

Usage

```
new_plugin(
  server,
  ui = NULL,
  validator = function(x, ...) x,
  class = character()
)

is_plugin(x)

as_plugin(x)

board_plugins(which = NULL)

plugins(...)

is_plugins(x)

as_plugins(x)

validate_plugins(x)
```

Arguments

`server`, `ui`	Server/UI for the plugin module
`validator`	Validator function that validates server return values
`class`	Plugin subclass
`x`	Plugin object
`which`	(Optional) character vectors of plugins to include
`...`	Plugin objects

Value

Constructors `new_plugin()`/`plugins()` return `plugin` and `plugins` objects, respectively, as do `as_plugin()`/`as_plugins()` and validators `validate_plugin()`/`validate_plugins()`, which are typically called for their side effects of throwing errors in case of validation failure. Inheritance checkers `is_plugin()`/`is_plugins()` return scalar logicals and finally, the convenience function `board_plugins()` returns a `plugins` object with all known plugins (or a selected subset thereof).

Examples

```
plg <- board_plugins()

is_plugins(plg)
names(plg)

plg[1:3]

is_plugin(plg[["preserve_board"]])
```

---

new_stack *Stacks*

Description

Multiple (related) blocks can be grouped together into stacks. Such a grouping has no functional implications, rather it is an organizational tool to help users manage more complex pipelines. Stack objects constitute a set of attributes, the most important of which is `blocks` (a character vector of block IDs). Each `stack` may have an arbitrary `name` and the class can be extended by adding further attributes, maybe something like `color`, coupled with sub-classing.

Stack container objects (`stacks` objects) can be created with `stacks()` or `as_stacks()` and inheritance can be tested via `is_stacks()`. Further basic operations such as concatenation, subsetting and sub-assignments is available by means of base R generics.

Usage

```
new_stack(blocks = character(), name = NULL, ..., class = character())

is_stack(x)

stack_blocks(x)

stack_blocks(x) <- value

stack_name(x, name)

stack_name(x) <- value
```

```
validate_stack(x)

as_stack(x)

stacks(...)

is_stacks(x)

as_stacks(x, ...)
```

Arguments

blocks	Set of blocks
name	Stack name
...	Extensibility
class	(Optional) stack sub-class
x	Stack object
value	Replacement value

Details

Individual stacks can be created using `new_stack()` or `as_stack()` and inheritance can be tested with `is_stack()`. Attributes can be retrieved (and modified) with `stack_blocks()`/`stack_blocks<-()` and `stack_name()`/`stack_name<-()`, while validation is available as (generic) `validate_stack()`.

Value

Construction and coercion via `new_stack()`/`as_stack()` and `stacks()`/`as_stacks()` results in `stack` and `stacks` objects, respectively, while inheritance testing via `is_stack()` and `is_stacks()` returns scalar logicals. Attribute getters `stack_name()` and `stack_blocks()` return scalar and vector-valued character vectors while setters `stack_name()<-` and `stack_blocks()<-` return modified stack objects.

Examples

```
stk <- new_stack(letters[1:5], "Alphabet 1")

stack_blocks(stk)
stack_name(stk)
stack_name(stk) <- "Alphabet start"

stks <- c(start = stk, cont = new_stack(letters[6:10], "Alphabet cont."))
names(stks)

tryCatch(
  stack_blocks(stks[[2]]) <- letters[4:8],
  error = function(e) conditionMessage(e)
)
```

new_transform_block *Transform block constructors*

Description

Many data transformations are be provided by blocks constructed via `new_transform_block()`, including examples where a single `data.frame` is transformed into another (e.g. `subset_block`), and two or more `data.frames` are combined (e.g. `merge_block` or `rbind_block`).

Usage

```
new_transform_block(server, ui, class, ctor = sys.parent(), ...)

new_head_block(n = 6L, direction = c("head", "tail"), ...)

new_merge_block(by = character(), all_x = FALSE, all_y = FALSE, ...)

new_rbind_block(...)

new_subset_block(subset = "", select = "", ...)
```

Arguments

`server`	A function returning `shiny::moduleServer()`
`ui`	A function with a single argument (`ns`) returning a `shiny.tag`
`class`	Block subclass
`ctor`	String-valued constructor name or function/frame number (mostly for internal use or when defining constructors for virtual classes)
`...`	Forwarded to `new_transform_block()` and `new_block()`
`n`	Number of rows
`direction`	Either "head" or "tail"
`by`	Column(s) tp join on
`all_x, all_y`	Join type, see `base::merge()`
`subset, select`	Expressions (passed as strings)

Value

All blocks constructed via `new_transform_block()` inherit from `transform_block`.

Head block

Row-subsetting the first or last `n` rows of a `data.frame` (as provided by `utils::head()` and `utils::tail()`) is implemented as `head_block`. This is an example of a block that takes a single `data.frame` as input and produces a single `data.frame` as output.

Merge block

Joining together two `data.frames`, based on a set of index columns, using `base::merge()` is available as `merge_block`. Depending on values passed as `all_x`/`all_y` the result will correspond to an "inner", "outer", "lfet" or "right" join. See `base::merge()` for details. This block class serves as an example for a transform block that takes exactly two data inputs `x` and `y` to produce a single `data.frame` as output.

Row-bind block

Row-wise concatenation of an arbitrary number of `data.frames`, as performed by `base::rbind()` is available as an `rbind_block`. This mainly serves as an example for a variadic block via the "special" `...args` block data argument.

Subset block

This block allows to perform row and column subsetting on `data.frame` objects via `base::subset()`. Using non-standard evaluation, strings passed as `subset`/`select` arguments or entered via shiny UI are turned into `language` objects by `base::parse()`.

`notify_user` *User notification plugin module*

Description

During the evaluation cycle of each block, user notifications may be generated to inform in case of issues such as errors or warnings. These notifications are provided in a way that display can be controlled and adapted to specific needs. The default `notify_user` plugin simply displays notifications via `shiny::showNotification()`, with some ID management in order to be able to clear no longer relevant notifications via `shiny::removeNotification()`.

Usage

```
notify_user(server = notify_user_server, ui = NULL)

notify_user_server(id, board, ...)
```

Arguments

`server`, `ui`	Server/UI for the plugin module
`id`	Namespace ID
`board`	Reactive values object
`...`	Extra arguments passed from parent scope

Value

A plugin container inheriting from `notify_user` is returned by `notify_user()`, while the UI component (e.g. `notify_user_ui()`) is expected to return shiny UI (i.e. `shiny::tagList()`; if available) and the server component (i.e. `notify_user_server()`) is expected to return a `shiny::reactiveVal()` or `shiny::reactive()` which evaluates to a list containing notifications per block and notification type (i.e. "message", "warning" or "error").

`preserve_board` *Serialization plugin module*

Description

Board state can be preserved by serializing all contained objects and restored via de-serialization. This mechanism can be used to power features such as save/restore (via download, as implemented in the default `preserve_board` plugin), but more refined user experience is conceivable in terms of undo/redo functionality and (automatic) saving of board state. Such enhancements can be implemented in a third-party `preserve_board` module.

Usage

```
preserve_board(server = preserve_board_server, ui = preserve_board_ui)

preserve_board_server(id, board, ...)

preserve_board_ui(id, board)
```

Arguments

`server, ui`	Server/UI for the plugin module
`id`	Namespace ID
`board`	The initial `board` object
`...`	Extra arguments passed from parent scope

Value

A plugin container inheriting from `preserve_board` is returned by `preserve_board()`, while the UI component (e.g. `preserve_board_ui()`) is expected to return shiny UI (i.e. `shiny::tagList()`) and the server component (i.e. `preserve_board_server()`) is expected to return a `shiny::reactiveVal()` or `shiny::reactive()` which evaluates to `NULL` or a `board` object.

`rand_names` *Random IDs*

Description

Randomly generated unique IDs are used throughout the package, created by `rand_names()`. If random strings are required that may not clash with a set of existing values, this can be guaranteed by passing them as `old_names`. The set of allowed characters can be controlled via `chars` and non-random pre- and suffixes may be specified as `prefix`/`suffix` arguments, while uniqueness is guaranteed including pre- and suffixes.

Usage

```
rand_names(
  old_names = character(0L),
  n = 1L,
  length = 15L,
  chars = letters,
  prefix = "",
  suffix = ""
)
```

Arguments

`old_names`	Disallowed IDs
`n`	Number of IDs to generate
`length`	ID length
`chars`	Allowed characters
`prefix`, `suffix`	ID pre-/suffix

Value

A character vector of length `n` where each entry contains `length` characters (all among `chars` and start/end with `prefix`/`suffix`), is guaranteed to be unique and not present among values passed as `old_names`.

Examples

```
rand_names(chars = c(letters, LETTERS, 0:9))
rand_names(length = 5L)
rand_names(n = 5L, prefix = "pre-", suffix = "-suf")
```

register_block *Block registry*

Description

Listing of blocks is available via a block registry, which associates a block constructor with metadata in order to provide a browsable block directory. Every constructor is identified by a unique ID (uid), which by default is generated from the class vector (first element). If the class vector is not provided during registration, an object is instantiated (by calling the constructor with arguments `ctor` and `ctor_pkg` only) to derive this information. Block constructors therefore should be callable without block- specific arguments.

Usage

```
register_block(
  ctor,
  name,
  description,
  classes = NULL,
  uid = NULL,
  category = "uncategorized",
  package = NULL,
  overwrite = FALSE
)

list_blocks()

unregister_blocks(uid = list_blocks())

register_blocks(...)

available_blocks()

create_block(id, ...)
```

Arguments

`ctor`	Block constructor
`name, description`	Metadata describing the block
`classes`	Block classes
`uid`	Unique ID for a registry entry
`category`	Useful to sort blocks by topics. If not specified, blocks are uncategorized.
`package`	Package where constructor is defined (or `NULL`)
`overwrite`	Overwrite existing entry

...	Forwarded to register_block()
id	Block ID as reported by list_blocks()

Details

Due to current requirements for serialization/deserialization, we keep track the constructor that was used for block instantiation. This works most reliable whenever a block constructor is an exported function from a package as this function is guaranteed to be available in a new session (give the package is installed in an appropriate version). While it is possible to register a block passing a "local" function as ctor, this may introduce failure modes that are less obvious (for example when such a constructor calls another function that is only defined within the scope of the session). It is therefore encouraged to only rely on exported function constructors. These can also be passed as strings and together with the value of package, the corresponding function can easily be retrieved in any session.

Blocks can be registered (i.e. added to the registry) via register_block() with scalar-valued arguments and register_blocks(), where arguments may be vector-valued, while de-registration (or removal) is handled via unregister_blocks(). A listing of all available blocks can be created as list_blocks(), which will return registry IDs and available_blocks(), which provides a set of (named) registry_entry objects. Finally, block construction via a registry ID is available as create_block().

Value

register_block() and register_blocks() are invoked for their side effects and return registry_entry object(s) invisibly, while unregister_blocks() returns NULL (invisibly). Listing via list_blocks() returns a character vector and a list of registry_entry object(s) for available_blocks(). Finally, create_block() returns a newly instantiated block object.

Examples

```
blks <- list_blocks()
register_block("new_dataset_block", "Test", "Registry test",
               uid = "test_block", package = "blockr.core")
new <- setdiff(list_blocks(), blks)
unregister_blocks(new)
setequal(list_blocks(), blks)
```

---

serve *Serve object*

Description

Intended as entry point to start up a shiny app, the generic function serve() can be dispatched either on a single block (mainly for previewing purposes during block development) or an entire board

Usage

```
serve(x, ...)

## S3 method for class 'block'
serve(x, id = "block", ..., data = list())

## S3 method for class 'board'
serve(x, id = rand_names(), plugins = board_plugins(), ...)
```

Arguments

x	Object
...	Generic consistency
id	Board namespace ID
data	Data inputs
plugins	Board plugins

Value

The generic `serve()` is expected to return the result of a call to `shiny::shinyApp()`.

Examples in Shinylive

example-1 Open in Shinylive

example-2 Open in Shinylive

`stack_ui` *Stack UI*

Description

Several generics are exported in order to integrate stack UI into board UI. We have `stack_ui()` which is dispatched on the `board` (and in the default implementation) on individual `stack` objects. This renders stacks as bootstrap accordion items (using `bslib::accordion()`). If a different way of displaying stacks and integrating them with a board is desired, this can be implemented by introducing a board subclass and providing a `stack_ui()` method for that subclass. Inserting stacks into (and removing stacks from) a board is available as `insert_stack_ui()`/`remove_stack_ui()` and blocks into/from stacks via `add_block_to_stack()`/`remove_block_from_stack()`. All are S3 generics with implementations for `board` and alternative implementation may be provided for board sub-classes.

Usage

```
stack_ui(id, x, ...)

## S3 method for class 'board'
stack_ui(id, x, stacks = NULL, edit_ui = NULL, ...)

## S3 method for class 'stack'
stack_ui(id, x, edit_ui = NULL, ...)

insert_stack_ui(
  id,
  x,
  board,
  edit_ui = NULL,
  session = getDefaultReactiveDomain(),
  ...
)

## S3 method for class 'board'
insert_stack_ui(
  id,
  x,
  board,
  edit_ui = NULL,
  session = getDefaultReactiveDomain(),
  ...
)

remove_stack_ui(id, board, session = getDefaultReactiveDomain(), ...)

## S3 method for class 'board'
remove_stack_ui(id, board, session = getDefaultReactiveDomain(), ...)

add_block_to_stack(
  board,
  block_id,
  stack_id,
  session = getDefaultReactiveDomain(),
  ...
)

## S3 method for class 'board'
add_block_to_stack(
  board,
  block_id,
  stack_id,
  session = getDefaultReactiveDomain(),
  ...
```

```
)

remove_block_from_stack(
  board,
  block_id,
  board_id,
  session = getDefaultReactiveDomain(),
  ...
)

## S3 method for class 'board'
remove_block_from_stack(
  board,
  block_id,
  board_id,
  session = getDefaultReactiveDomain(),
  ...
)
```

Arguments

id	Parent namespace
x	Object
...	Generic consistency
stacks	(Additional) stacks (or IDs) for which to generate the UI
edit_ui	Stack edit plugin
board	Board object
session	Shiny session
block_id, stack_id, board_id	Block/stack/board IDs

Value

UI set up via `stack_ui()` is expected to return `shiny::tag()` or `shiny::tagList()` objects while stack/block insertion/removal functions (into/from board/stack objects) are called for their side-effects. Both `insert_stack_ui()`/`remove_stack_ui` and `add_block_to_stack()`/`remove_block_from_stack()` return `NULL` invisibly and where the former call `shiny::insertUI()`/`shiny::removeUI()` and the latter modify the DOM via shiny::session custom messages.

`write_log` *Logging*

Description

Internally used infrastructure for emitting log messages is exported, hoping that other packages which depend on this, use it and thereby logging is carried out consistently both in terms of presentation and output device. All log messages are associated with an (ordered) level ("fatal", "error", "warn", "info", "debug" or "trace") which is compared against the currently set value (available as `get_log_level()`) and output is only generated if the message level is greater or equal to the currently set value.

Usage

```
write_log(..., level = "info")

log_fatal(...)

log_error(...)

log_warn(...)

log_info(...)

log_debug(...)

log_trace(...)

as_log_level(level)

get_log_level()

cnd_logger(msg, level)

cat_logger(msg, level)
```

Arguments

`...`	Concatenated as `paste0(..., "\n")`
`level`	Logging level (possible values are "fatal", "error", "warn", "info", "debug" and "trace"
`msg`	Message (string)

Value

Logging function `write_log()`, wrappers `log_*()` and loggers provided as `cnd_logger()`/cat_logger() all return `NULL` invisibly and are called for their side effect of emitting a message. Helpers `as_log_level()` and `get_log_level()` return a scalar-valued ordered factor.

Index