Package 'blockr.core'

July 12, 2026

Title Graphical Web-Framework for Data Manipulation and Visualization

Version 0.1.3

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

Imports shiny (>= 1.5.0), DT, bslib, bsicons, datasets, grDevices, graphics, methods, utils, jsonlite, vctrs, generics, rlang, htmltools, evaluate, shinyFiles, digest, cli, glue, yaml

Suggests testthat (>= 3.0.0), memuse, withr, shinytest2, chromote, roxy.shinylive, roxygen2, knitr, rmarkdown, quarto, scoutbaR, thematic, ids, ellmer

Config/testthat/edition 3

VignetteBuilder quarto

Collate 'block-args.R' 'block-class.R' 'block-eval.R' 'block-meta.R' 'block-registry.R' 'block-roclet.R' 'block-server.R' 'block-ui.R' 'blocks-class.R' 'board-class.R' 'board-loader.R' 'board-lock.R' 'board-option.R' 'board-options.R' 'board-plugins.R' 'board-server.R' 'board-ui.R' 'utils-dt.R' 'data-block.R' 'data-dataset.R' 'data-static.R' 'file-block.R' 'file-browser.R' 'file-upload.R' 'link-class.R' 'links-class.R' 'parser-block.R' 'parser-csv.R' 'plot-block.R' 'plot-scatter.R' 'plugin-block.R' 'plugin-blocks.R' 'plugin-code.R' 'plugin-control.R' 'plugin-links.R' 'plugin-notification.R' 'plugin-serdes.R' 'plugin-stack.R' 'plugin-stacks.R' 'reactives.R' 'stack-class.R' 'stack-ui.R' 'stacks-class.R' 'str-value.R' 'text-block.R' 'text-glue.R' 'transform-block.R' 'transform-fixed.R' 'transform-head.R' 'transform-merge.R' 'transform-rbind.R' 'transform-subset.R' 'utils-assertions.R'

'utils-cnd.R' 'utils-code.R' 'utils-expr.R' 'utils-graph.R'
'utils-logging.R' 'utils-misc.R' 'utils-pkg.R' 'utils-ply.R'
'utils-serdes.R' 'utils-serve.R' 'utils-shiny.R'
'utils-tests.R' 'zzz-onload.R'

Config/roxygen2/version 8.0.0

NeedsCompilation no

Author Nicolas Bennett [aut, cre],
David Granjon [aut],
Christoph Sax [aut],
Bristol Myers Squibb [fnd]

Maintainer Nicolas Bennett <nicolas@cynkra.com>

Repository CRAN

Date/Publication 2026-07-12 21:00:02 UTC

Contents

`bbquote` *Quoting utilities*

Description

Block expressions in blockr are evaluated in a 2-step manner: first in the context of arguments supplied by the user via UI elements and in a second step in the context of input data. The function `base::bquote()` does not allow for terms wrapped in `.()` or `..()` to be missing and this makes it incompatible with this 2-step approach. A drop-in replacement, provided as `bbquote()` addresses this shortcoming.

Usage

```
bbquote(expr, where = parent.frame(), splice = FALSE)

.(x)

..(x)
```

Arguments

`expr`	A language object.
`where`	An environment.
`splice`	Logical; if `TRUE` splicing is enabled.
`x`	Object

Details

A block like `new_head_block()` is expected to return an expression of the form `utils::head(data, n = 10L)`, which will then be evaluated in an environment where he have a name `data` bound to some dataset. In order to perform some manipulations of such block expressions it is required to somehow mark the terms that correspond to input data and for that we can use the syntax introduced by `base::bquote()`. What we would prefer to have as block expression therefore is not the above, but something like `utils::head(.(data), n = 10L)`, as this affords us more possibilities for performing substitutions (and therefore generates cleaner code).

In order to interpolate certain arguments in a first step, we unfortunately cannot use `base::bquote()`, but we can use `bbquote()` instead to generate the desired expression.

```
bquote(utils::head(.(data), n = .(n)), list(n = 10L))
#> Error in eval(e[[2L]], where) : object 'data' not found
bbquote(utils::head(.(data), n = .(n)), list(n = 10L))
#> utils::head(.(data), n = 10L)
```

This also works with `..()` and splicing.

Value

A language object in the same way as returned by `base::bquote()`. Functions `.()` and `..()` throw errors when invoked an only exist to mask check notes "no visible global function definition" for their use.

Examples

```
bbquote(utils::head(.(data), n = .(n)), list(n = 10L))
bbquote(c(.(a), ..(bc)), list(a = "a"))
bbquote(c(.(a), ..(bc)), list(a = "a", bc = c("b", "c")), splice = TRUE)
```

`blockr_abort` *Blockr conditions*

Description

Wrappers for `rlang::abort()`, `rlang::warn()` and `rlang::inform()`. In addition to `class`, conditions inherit from "blockr_error".

Usage

```
blockr_abort(..., class = character(), envir = parent.frame())

blockr_warn(
  ...,
  class = character(),
  envir = parent.frame(),
```

```
  frequency = "always",
  frequency_id = NULL
)

blockr_inform(
  ...,
  class = character(),
  envir = parent.frame(),
  frequency = "always",
  frequency_id = NULL
)
```

Arguments

`...`	Forwarded to `cli::pluralize()`
`class`	Condition class
`envir`	Forwarded to `cli::pluralize()`
`frequency, frequency_id`	Forwarded to `rlang::warn()`

Value

Called for side-effect of signaling conditions.

`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)

set_blockr_options(...)
```

Arguments

`name`	Option name
`default`	Default value
`...`	Option key value pairs as named arguments

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

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 'blockr_ctor'
blockr_ser(x, ...)
```

```
## S3 method for class 'board_option'
blockr_ser(x, option = NULL, ...)

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

## S3 method for class 'board'
blockr_ser(x, board_id = 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, ...)

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

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

Arguments

x	Object to (de)serialize
...	Generic consistency
state	Object state (as returned from an expr_server)
blocks	Block states (NULL, or a per-block NULL entry, defaults to values from the constructor scope)
options	Board option values (NULL uses values provided by x)
option	Board option value (NULL uses values provided by x)
board_id	Board ID
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).

During deserialization, blockr_deser() forwards ... to the dispatched per-class method. This lets callers (and outer methods deserializing nested objects) thread additional context down to inner deserializers.

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.

Examples

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

blockr_ser(blk)

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

block_eval *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_eval(x, expr, env, ...)

eval_env(data)

block_eval_trigger(x, session = get_session())

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

## S3 method for class 'block'
block_server(
  id,
  x,
  data = list(),
  block_id = id,
  edit_block = NULL,
  ctrl_block = NULL,
  board = reactiveValues(),
  update = reactiveVal(),
  inputs_ready = reactive(TRUE),
  ...
)

expr_server(x, data, ...)

block_render_trigger(x, session = get_session())
```

Arguments

`x`	Object for which to generate a `shiny::moduleServer()`
`expr`	Quoted expression to evaluate in the context of `data`
`env`	Environment in which to evaluate `expr`
`...`	Generic consistency

`data`	Input data (list of reactives)
`session`	Shiny session object
`id`	Namespace ID
`block_id`	Block ID
`edit_block, ctrl_block`	Block plugins
`board`	Reactive values object containing board information
`update`	Reactive value object to initiate board updates
`inputs_ready`	Reactive flag signaling whether the block's required inputs are all connected to ready upstream blocks (supplied by `board_server()`; defaults to always-ready when a block server is run standalone)

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).

Each block carries an *eval status* – one of `dormant`, `waiting`, `unset`, `failed` or `ready` – which, together with the orthogonal `visible` flag, determines its behaviour. The status separates the two input kinds (data inputs from links, user inputs from `state`) and a genuine failure:

- `dormant` – not *needed* (neither on screen nor feeding, transitively over `board_links()`, an on-screen block); inputs stay unfulfilled (`shiny::req()` out) and nothing evaluates.
- `waiting` – needed, but a required *data* input is missing: unconnected, below the required number of variadic `...args` inputs (one by default), or fed by an upstream block that is not itself `ready` (see `allow_empty_state`).
- `unset` – data inputs are ready, but a required *user* input (`state` value) has not been provided (unless permitted by `allow_empty_state`).
- `failed` – all inputs are present, but the block cannot produce a result: the data validator (`validate_data_inputs()`) or the block expression raised. The offending condition is surfaced through the block conditions.
- `ready` – evaluation succeeded and a result (possibly a legitimate `NULL`) is available for downstream blocks to consume.

A block reaches `ready` only once its upstreams have, so an unconnected or pending block holds its whole downstream chain `waiting` without any of them evaluating against missing data. Output rendering follows the status: the block output is shown only while `ready` and cleared otherwise, so a block leaving `ready` never displays a stale result. While not `ready` the block surfaces a condition explaining why – a `status`-phase note for `waiting` and `unset`, or the raised error for `failed`. Conditions raised during validation and evaluation are caught and returned 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.

When a front-end (such as blockr.dock) drives the `visible` write-channel that `board_server()` hands to the board callback, reporting each block's visibility status (off screen, on screen, or rendered into its view), evaluation and rendering are gated on visibility. Rendering is gated on plain visibility: the render observer is suspended while a block is off screen and resumed once it is on screen, starting suspended so nothing renders before the front-end first reports. Evaluation is gated on the *needed* set, the on-screen blocks together with their upstream closure over `board_links()` (derived from `visible`, recomputed only when it or the links change). A block's input data reactives stay unfulfilled (they `shiny::req()` out) unless the block is needed, so a block that is neither visible nor feeding a visible block pulls no input and stays fully quiescent: its result reactive, and any observer its expression server registers on the incoming data, all short-circuit and do nothing. A needed but off-screen block (one feeding a visible block) evaluates but does not render. Block-server *construction* is prioritized the same way: the needed set is instantiated first so that first paint waits only for the on-screen blocks and their upstreams, and the remaining block servers are built progressively in the background. That background pass holds until the front-end reports every on-screen block as rendered (arranged into its view), so it never competes with first paint. Until a block is built it is absent from the `board$blocks` handed to plugins and callbacks, which simply see it appear once constructed. The background cadence is set by the `background_construction_delay` `blockr_option()` (milliseconds between successive blocks, default 50); a value of 0 disables the staggering and builds every block up front. With nothing driving `visible` every block is needed and behaviour is unchanged; the `gate_visibility` `blockr_option()` (default `TRUE`) turns gating off entirely.

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_metadata` *Block metadata*

Description

Registry metadata for blocks is available both as a tabular overview and via per-attribute accessors. `block_metadata()` returns a `data.frame` with one row per block – dispatching on a `block`, a `blocks` collection, a `block_registry_entry` or a registry ID – where scalar attributes are atomic columns and the multi-valued ones (`arguments`, `examples`, `keywords`) are list-columns. The `fields` argument selects a subset of columns. Each attribute additionally has a dedicated getter (`block_meta_name()`, `block_meta_guidance()`, ...) returning that attribute for a single block. Missing fields are filled with display defaults in the data.frame; the getters instead return the stored value (or `NA` / an empty value).

Usage

```
block_metadata(x, fields = "all", ...)

block_meta_id(x, ...)
```

```
block_meta_name(x, ...)

block_meta_description(x, ...)

block_meta_details(x, ...)

block_meta_link(x, ...)

block_meta_guidance(x, ...)

block_meta_category(x, ...)

block_meta_icon(x, ...)

block_meta_package(x, ...)

block_meta_keywords(x, ...)

block_meta_arguments(x, ...)

block_meta_examples(x, ...)
```

Arguments

x	A block, a blocks collection, a block_registry_entry or a registry ID
fields	Metadata fields to include (defaults to "all")
...	Generic consistency, passed on to methods

Value

block_metadata() returns a data.frame. The block_meta_*() getters return the named attribute: a string (or NA) for scalar fields, a character vector for block_meta_keywords(), a block_args object for block_meta_arguments(), and a list of worked configurations for block_meta_examples().

---

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)

external_ctrl_vars(x)

has_external_ctrl(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,
- external_ctrl_vars(): a character vector of externally controllable variable names (always including "block_name" for blocks),
- has_external_ctrl(): a scalar logical.

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()`.

External control

Blocks can expose constructor inputs for programmatic control from outside the block server (see the `external_ctrl` argument to `new_block()` and the `ctrl_block()` plugin). `external_ctrl_vars()` is a generic that resolves this declaration into the concrete set of controllable variable names: `TRUE` expands to all constructor inputs, `FALSE` to none and a character vector is taken as a (validated) subset. For blocks, `"block_name"` is always included as every block can be renamed. The predicate `has_external_ctrl()` reports whether this set is non-empty (always `TRUE` for blocks).

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())

external_ctrl_vars(new_dataset_block())
has_external_ctrl(new_dataset_block())
```

`block_roclet` *Block registration roclet*

Description

A custom roxygen2 roclet that turns block registration metadata, declared as tags on block constructors, into a YAML registry (`inst/registry/blocks.yml`) that `register_package_blocks()` reads at load time. It runs alongside the standard roclets during `roxygen2::roxygenise()` (or `devtools::document()`) for any package that lists `blockr.core::block_registration_roclet` in the `Roxygen` field of its `DESCRIPTION`.

Usage

```
block_registration_roclet()
```

Value

`block_registration_roclet()` returns a roclet object.

Tags

Placed in the roxygen block above a block constructor:

`@block <name>` Human-readable block name. Required: its presence is what marks a constructor for registration.

`@blockDescr <text>` Short block description. Required.

`@blockCategory <category>` One of the `suggested_categories()`. Required.

`@blockIcon <icon>` Bootstrap icon name. Optional, defaulting to the category icon (see `default_icon()`).

`@blockGuidance <text>` Model-facing construction guidance – do and don't rules, enumerations, pitfalls. Optional.

`@blockKeywords <terms>` Comma-separated search terms for block discovery; a term may contain spaces. Optional.

`@blockDetails <text>` Longer human-facing description (e.g. for a help popover). Optional; when omitted it falls back to the constructor's `@details` or, failing that, its `@section` prose.

`@blockLink <url>` URL of the block's help or documentation page. Optional; when omitted it is derived from the package's pkgdown `url` (`_pkgdown.yml` or `DESCRIPTION`) and the documented topic, as `<url>/reference/<topic>.html`.

`@blockArg <name> <description>` One constructor argument's specification. The text after the name is a free-text description; `[example] <expr>` and `[type] <expr>` markers (each on its own line) supply an R expression – evaluated when documentation is generated – for a worked example value and an `arg_*()` type descriptor (see `new_block_arg()`). An explicit `[description] <text>` marker may replace the inline description. Optional and repeatable.

`@blockExamples <expr>` Block-level worked example configurations, as an R expression – evaluated when documentation is generated – that yields a list of complete configurations (each a named list keyed by argument). Optional; supersedes the per-argument `[example]` assembly.

`@blockCtor <name>` Constructor name override. Optional: the documented object otherwise supplies the name.

Multi-argument worked `examples` (whole-block configurations, as opposed to a per-argument `[example]`) remain out of the tags' scope; a block needing those registers via `register_block()` directly.

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, ctrl_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, ctrl_ui`	Block plugin UI

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, ..., session = get_session())

board_links(x)

board_links(x) <- value

board_link_ids(x)

modify_board_links(
  x,
  add = NULL,
  rm = NULL,
  mod = NULL,
  ...,
  session = get_session()
```

```
)

board_stacks(x)

board_stacks(x) <- value

board_stack_ids(x)

modify_board_stacks(
  x,
  add = NULL,
  rm = NULL,
  mod = NULL,
  ...,
  session = get_session()
)

board_options(x)

board_options(x) <- value

board_option_ids(x)

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

clear_board(x)
```

Arguments

x	Board
value	Replacement value
rm	Block/link/stack IDs to remove
...	Further arguments they may be passed from the board server context
session	Shiny session object
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()/board_optio
and board_blocks<-()/board_links<-()/board_stacks<-()/board_options<-()) return corresponding objects (i.e. blocks, links, stacks and board_options), while ID getters (board_block_ids(),

board_link_ids(), board_stack_ids() and board_option_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().

Options

Board options can be retrieved using board_options() and updated with the corresponding replacement function board_options<-(). If only the option IDs are of interest, this is available as board_option_ids(), which calls board_option_id() on each board option.

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(brd)
```

---

board_ctor *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().

Usage

```
board_ctor(x)

new_board_option(
  id,
  default,
  ui,
  server = function(board, ..., session) {
 },
  update_trigger = id,
  transform = identity,
  category = NULL,
  ctor = sys.parent(),
  pkg = NULL
)

is_board_option(x)

validate_board_option(x)

as_board_option(x, ...)

## S3 method for class 'board_option'
as_board_option(x, ...)

board_option_id(x)
```

```
board_option_trigger(x)

board_option_default(x)

board_option_category(x)

board_option_ui(x, id = NULL)

board_option_server(x, ...)

board_option_transform(x)

board_option_value(x, value = board_option_default(x))

board_option_ctor(x)

## Default S3 method:
validate_board_option(x)

new_board_name_option(value = NULL, category = "Board options", ...)

new_n_rows_option(
  value = blockr_option("n_rows", 50L),
  category = "Table options",
  ...
)

new_page_size_option(
  value = blockr_option("page_size", 5L),
  category = "Table options",
  ...
)

new_filter_rows_option(
  value = blockr_option("filter_rows", FALSE),
  category = "Table options",
  ...
)

new_thematic_option(
  value = blockr_option("thematic", NULL),
  category = "Theme options",
  ...
)

new_dark_mode_option(
  value = blockr_option("dark_mode", NULL),
  category = "Theme options",
```

```
  ...
)

new_show_conditions_option(
  value = blockr_option("show_conditions", c("warning", "error")),
  category = "Board options",
  ...
)

new_llm_model_option(value = NULL, category = "Board options", ...)

new_board_options(...)

default_board_options(...)

is_board_options(x)

as_board_options(x)

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

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

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

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

validate_board_options(x)

board_option_values(x)

get_board_option_value(opt, session = get_session())

set_board_option_value(opt, val, board, session = get_session())

get_board_option_or_default(
  opt,
  opts = default_board_options(),
  session = get_session()
)

get_board_option_or_null(opt, session = get_session())

get_board_option_values(
```

```
  ...,
  opts = default_board_options(),
  if_not_found = c("error", "default", "null"),
  session = get_session()
)

combine_board_options(...)
```

Arguments

x	Board options object
id	Board option ID
default	Default value
ui	Option UI
server	(Optional) option server
update_trigger	Shiny input entry/entries that trigger an update
transform	(Optional) transform function
category	(Optional) string-valued category
ctor, pkg	Constructor information (used for serialization)
...	Options passed as individual arguments
value	Option value
opt	Option name
session	Shiny session
val	New value
board	Board the option belongs to, used to resolve the lock state (see is_board_locked()).
opts	Board options
if_not_found	Behavior in case an option is not found

Value

All of new_board_options() and as_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 board_option_values() returns a named list of option values.

Examples

```
opt <- new_board_options(
   new_board_name_option(),
   new_page_size_option()
 )

is_board_options(opt)
names(opt)

opt[["page_size"]]
```

board_loader *Board loader*

Description

Which board to build for an incoming request – and how a board is carried across the session$reload() that a preserve_board restore triggers – is the job of an app-level **board loader**, passed to serve() as its loader argument. A board_loader() pairs a resolve(request, session, default) – returning the board to build for an incoming request, or NULL for the serve() default – with an optional stage(board, session), which persists a board and returns the URL query parameters that reference it (a resolve-only loader, e.g. one not backing a restore, leaves stage NULL). serve() uses that one loader for both the request-phase resolution (at the GET, where session is NULL, and at the WS connect) and the in-session staging when a restore fires; **core** writes those parameters into the URL and drives the reload, so the reload stays a guaranteed core mechanism that no loader can opt out of.

Usage

```
board_loader(resolve, stage = NULL)

is_board_loader(x)

local_loader()
```

Arguments

resolve, stage	Paired functions backing a board_loader: resolve is function(request, session, default) returning the board to build or NULL; stage is function(board, session) (or NULL for a loader that does not stage, e.g. resolve-only) which persists board and returns the URL query parameters referencing it (core writes them and reloads). They share private state, so a board_loader is built as a unit, not supplied as two loose functions.
x	Object to test for board_loader-ness

Details

resolve receives the raw HTTP request at both phases (the UI request at the GET, session$request at the WS), the session (NULL at the GET), and default – the board passed to serve() (the serve() default, built when resolve returns NULL), which a loader can also derive its own result from (e.g. clear_board(default)). A loader that keys off URL query parameters reads them itself, minding the phase split: the query is on request$QUERY_STRING at the GET but session$clientData$url_search at the WS (the websocket request carries neither).

The default local_loader() keeps its handoff in a per-loader store (no process global) and is therefore single-process; multi-user deployments pass a loader resolving from a shared backend (as blockr.session does).

Value

board_loader() and local_loader() return a board_loader object and is_board_loader() a scalar logical.

---

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 = board_plugins(x),
  options = board_options(x),
  callbacks = list(),
  callback_location = c("end", "start"),
  ...
)
```

Arguments

id	Parent namespace
x	Board
...	Generic consistency
plugins	Board plugins as modules
options	Board options (NULL defaults to the union of board, block and registry sourced options)
callbacks	Single (or list of) callback function(s), called only for their side-effects)
callback_location	Location of callback invocation (before or after plugins)

Value

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

Active conditions

Conditions raised while blocks evaluate (errors, warnings and messages) are exposed as a reactive data frame `board$conditions` on the read-only board handed to plugins and callbacks, with one row per active condition and columns `block`, `phase`, `severity`, `message` and `id`. It combines the per-block `server$conditions` reactives (see `block_server()`), so a consumer reads a single reactive — the whole board, or one block's frame for fine-grained updates — rather than walking nested condition state. The default `notify_user()` plugin renders its toasts from this source.

`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. Additionally, `toolbar_ui()` is responsible for creating a toolbar UI component from several plugin UI components.

Usage

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

board_ui(id, x, ...)

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

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

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

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

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

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

toolbar_ui(id, x, plugins = list(), ...)
```

```
## S3 method for class 'board'
toolbar_ui(id, x, plugins = list(), options = NULL, ...)
```

Arguments

id	Namespace ID
x	Board
...	Generic consistency
plugins	UI for board plugins
options	Board options (NULL defaults to the union of board, block and registry sourced options)
blocks	(Additional) blocks (or IDs) for which to generate the UI
session	Shiny session

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. 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, as does toolbar_ui(), while updater functions (insert_block_ui() and remove_block_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.

---

board_update *Board update*

Description

Inside board_server() every state change flows through one board_update reactive. Core registers two observers framing the change: an initial one that validates the payload and runs augment_board_update() for auto-fixups, and a final one that runs apply_board_update() and resets the reactive. Plugins or callbacks may register their own observers in between, provided they use a *finite* priority — the highest and lowest reactive priorities are reserved for core.

Usage

```
validate_board_update(payload, board, ..., session = get_session())

augment_board_update(upd, board, ..., session = get_session())

apply_board_update(board, upd, ..., session = get_session())
```

Arguments

payload, upd	A board update payload — see Validation above for the accepted shape.
board	A board object.
...	Forwarded between methods. For apply_board_update(), the final observer also splices board_server()'s ... in here.
session	A shiny session, default get_session().

Details

All three functions dispatch on the board class. Subclasses override to validate, augment, or react to their own payload slots, typically composing with NextMethod(). validate_board_update() is also a caller-facing entry point: it mirrors the initial observer's checks against a caller-supplied payload, useful for staging layers (e.g. accumulating LLM-proposed updates) that need to fail loudly before publishing.

Value

validate_board_update() returns invisible(payload) (or throws a blockr_abort() error). augment_board_update() returns the (possibly extended) payload. apply_board_update() returns a board.

Validation

The default .board method runs a structural check on the payload (block / link / stack per-slot rules) and a cross-reference check that link endpoints and stack members resolve in the post-update merged view. Unknown top-level keys are passed through, so subclass payload slots reach subclass augment / apply methods.

Augment

The default .board method inserts implied link removals and stack updates that follow from block removals, plus link-input completion. Subclass methods may extend the payload with their own fixups; an error thrown here aborts the update before apply runs.

Apply

The default .board method returns the supplied board unchanged — the core apply path (block / link / stack mutation, block UI insertion / removal) is not routed through this generic. Subclass methods receive a plain board snapshot (no reactive surface) and return a board, which the final observer assigns back to rv$board. For piecemeal customization of the core apply path itself, override the relevant sub-generic (modify_board_links(), insert_block_ui(), etc.) instead.

Errors thrown from either augment or apply are caught by the observer, reported via notify(), and the reactive is reset so the app keeps running.

Outcome

Alongside the human-facing notify() toast, every update cycle records a machine-readable result into board$last_update (the read-only board handed to plugins and callbacks). It is a list with a monotonically increasing seq, a logical ok, the phase it ended in ("validate" or "apply"), and a message (conditionMessage() on failure, NA on success); it is NULL before the first update. The seq advances on every write so that two consecutive identical outcomes still invalidate a downstream observer. A programmatic caller can watch this field to learn whether a dispatched update was rejected, failed to apply, or landed.

See Also

On a locked board (see is_board_locked()) the update is dropped rather than applied.

Examples

```
brd <- new_board(
  blocks = c(a = new_dataset_block("iris"), b = new_subset_block()),
  links = links(ab = new_link(from = "a", to = "b"))
)

validate_board_update(
  list(links = list(rm = "ab")),
  brd
)

try(
  validate_board_update(
    list(links = list(add = links(xy = new_link(from = "x", to = "y")))),
    brd
  )
)
```

---

ctrl_block *Plugin module for external control of block inputs*

Description

This plugin enables setting block reactive state values from outside the block expression server context. Blocks opt in to external control via the external_ctrl argument to new_block(), which can be set to TRUE (all constructor inputs) or a character vector of specific input names. The default UI renders a shiny::textInput() for each externally controllable input along with a submit shiny::actionButton(). Both the server and UI can be replaced with custom implementations by passing alternate functions to ctrl_block().

Usage

```
ctrl_block(server = ctrl_block_server, ui = ctrl_block_ui)

ctrl_block_server(id, x, vars, data, eval)

ctrl_block_ui(id, x)
```

Arguments

server, ui	Server/UI for the plugin module
id	Namespace ID
x	Block
vars	Reactive state values (list of reactiveVal objects keyed by input name). block_name is included by the default block_server.block even though it is not part of the block's expr_server state.
data	Input data paseed as list of reactive values
eval	Reactive that evaluates the block expression against input data. May be used to validate that the new values produce a successful evaluation.

Details

The default server validates submitted values by evaluating the block expression (via the eval reactive) after updating state. On success, a reactive gate is returned as TRUE, allowing downstream evaluation to proceed. On failure, state values are reverted to their previous values, the user is notified, and the gate is set to FALSE, which blocks downstream evaluation until a subsequent successful submit.

Value

A plugin container inheriting from ctrl_block is returned by ctrl_block(), while the UI component (i.e. ctrl_block_ui()) is expected to return shiny UI (i.e. shiny::tagList()) and the server component (i.e. ctrl_block_server()) is expected to return a value that passes validation (i.e. TRUE or a reactive gate).

---

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,
  validator = abort_not_null
)

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
validator	Validator function that validates server return values
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`.

---

export_code *Utilities for code export*

Description

To facilitate other means of code export than implemented by the default `generate_code()` plugin, this utility performs much of the heavy lifting to properly arrange and scope block-level expressions.

Usage

```
export_code(expressions, board)
```

Arguments

expressions	Block expressions
board	Board object

Value

String containing properly arranged block expressions.

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`	Reactive values 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`.

generate_plugin_args *Testing utilities*

Description

Several utilities for unit testing, mainly with `shiny::testServer()` that have proven themselves useful for testing this package are exported for re-use in other packages.

Usage

```
generate_plugin_args(board, ..., mode = c("edit", "read"))

sink_msg(...)

new_mock_session()

with_mock_session(expr, session = new_mock_session())

with_mock_context(session, expr)

get_s3_method(generic, object)

export_safely(x)
```

Arguments

board	A board object
...	Forwarded to utils::capture.output()
mode	Edit plugins, such as manage_blocks get an additional argument update over read plugins such as preserve_board.
expr	Test code containing expectations. The objects from inside the server function environment will be made available in the environment of the test expression (this is done using a data mask with rlang::eval_tidy()). This includes the parameters of the server function (e.g. input, output, and session), along with any other values created inside of the server function.
session	The MockShinySession object to use as the reactive domain. The same session object is used as the domain both during invocation of the server or module under test and during evaluation of expr.
generic	Generic function name (passed as string)
object	S3 Object
x	Reactive object to use in shiny::exportTestValues()

Value

For testing plugins, generate_plugin_args() returns objects that mimic how plugins are called in the board server (with reactive components re-created on a NULL reactive domain so they outlive the shiny::testServer() session used to set them up), sink_msg() is called mainly for the side-effect of muting shiny messages (and returns them invisibly), with_mock_session() returns NULL (invisibly) and with_mock_context() returns the result of a call to shiny::withReactiveDomain(). Finally, get_s3_method() returns a class-specific implementation of the specified generic (and throws an error if none is found).

get_session *Shiny utilities*

Description

Utility functions for shiny:

- get_session: See shiny::getDefaultReactiveDomain().
- generate_plugin_args: Meant for unit testing plugins.
- notify: Glue-capable wrapper for shiny::showNotification().

Usage

```
get_session()

notify(
  ...,
  envir = parent.frame(),
  action = NULL,
  duration = 5,
  close_button = TRUE,
  id = NULL,
  type = c("message", "warning", "error"),
  glue = TRUE,
  log = TRUE,
  session = get_session()
)
```

Arguments

...	Concatenated as paste0(..., "\n")
envir	Environment where the logging call originated from
action	Message content that represents an action. For example, this could be a link that the user can click on. This is separate from ui so customized layouts can handle the main notification content separately from action content.
duration	Number of seconds to display the message before it disappears. Use NULL to make the message not automatically disappear.
close_button	Passed as closeButton to shiny::showNotification()
id	A unique identifier for the notification. id is optional for showNotification(): Shiny will automatically create one if needed. If you do supply it, Shiny will update an existing notification if it exists, otherwise it will create a new one. id is required for removeNotification().
type	A string which controls the color of the notification. One of "default" (gray), "message" (blue), "warning" (yellow), or "error" (red).

glue, log	Whether to glue::glue()-interpolate ... and whether to emit a log message. Set both to FALSE to surface pre-formatted, already-logged text (e.g. a captured condition message, which may contain braces that would otherwise fail interpolation).
session	Session object to send notification to.

Value

Either NULL or a shiny session object for get_session(), a list of arguments for plugin server functions in the case of generate_plugin_args() and notify() is called for the side-effect of displaying a browser notification (and returns NULL invisibly).

---

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)
```

`is_scalar` *Assertions*

Description

Utility functions, mainly intended for asserting common preconditions are exported for convenience in dependent packages.

Usage

```
is_scalar(x)

is_string(x)

is_bool(x)

is_intish(x)

is_count(x, allow_zero = TRUE)

is_number(x)

not_null(...)

has_length(x)
```

Arguments

`x`	Object to check
`allow_zero`	Determines whether the value 0 is considered a valid count
`...`	Silently ignored

Value

Scalar logical value.

`locked-board` *Locked boards*

Description

A board can be deployed read-only by setting the `blockr.locked` option (see `blockr_option()`). Locking is enforced server-side, not by UI hiding: a forged `Shiny.setInputValue` bypasses any `display: none` and fires the underlying observer, so the refusal has to happen on the server. While locked, the two channels every mutation funnels through – the `board_update` lifecycle (see `board_update()`) and `set_board_option_value()` – refuse to apply changes.

Usage

```
is_board_locked(board, ...)

## S3 method for class 'board'
is_board_locked(board, ...)
```

Arguments

`board`	A `board` object to dispatch on.
`...`	Generic consistency.

Details

`is_board_locked()` reports whether a board is locked and dispatches on the board, so a subclass can source the state however it likes (for example gating an unlock behind an authenticated request). The default `board` method reads the `blockr.locked` option, and `blockr.dock`'s `is_dock_locked()` delegates here.

Value

A logical flag.

Scope

Locking guards against forged *client input* – a browser that sets Shiny inputs but cannot run server-side R. It is **not** a defense against code executing in the session: blockr evaluates arbitrary R, which can flip the option, rebind this generic or mutate board state directly, and no in-process flag can prevent that. A subclass method may raise the bar (e.g. an authenticated unlock) but cannot close the gap on its own – a hard boundary has to live where the user's code does not run (a sandboxed evaluator, or an ephemeral read-only deployment). Treat locking as a guard for non-malicious users.

Examples

```
is_board_locked(new_board())
```

`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 = sys.parent(),
  ctor_pkg = NULL,
  dat_valid = NULL,
  allow_empty_state = FALSE,
  block_name = default_block_name,
  expr_type = c("quoted", "bquoted"),
  external_ctrl = FALSE,
  block_metadata = NULL,
  ...
)

default_block_name(class)

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	Relaxes the block's readiness requirements. By default every data input must be connected and a variadic block needs at least one ...args input. As TRUE, FALSE or a character vector of state names it relaxes *user* inputs (which state values may be empty). A list(input = ..., data = ...) also relaxes *data* inputs: input takes the same TRUE/FALSE/character form, while data names non-variadic data inputs that may stay unconnected and, via a ...args entry, overrides the required number of variadic inputs, e.g. list(input = "n", data = list("y", ...args = 2))
block_name	Block name
expr_type	Expression type (experimental)
external_ctrl	Set up external control (experimental)
block_metadata	Block metadata
...	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 from blocks and is_blocks() 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 container object as `...args` and thus such a variadic block needs `...args` as part of the server function signature. The container mirrors `shiny::reactiveValues()` for reads (`names()`, `[[`, `$` and `as.list()`), and accessing a slot yields the value of its (lazily evaluated) input. The fixed per-data input arguments are instead 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.

Ahead of validation, a block must have its inputs available: every required *data* input connected to a ready upstream (variadic blocks need at least the `...args` minimum declared via `allow_empty_state`) and every required *user* input (`state`) provided. Until then – including while an upstream is itself still pending – neither the validator nor the block expression run and no output is produced. See `block_server()` for the resulting eval status (`dormant` / `waiting` / `unset` / `failed` / `ready`).

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) {
      htmltools::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_block_arg *Block argument specification*

Description

Block constructor arguments can be documented with a structured specification: each argument via `new_block_arg()` (a `description`, a single worked `example`, and an optional machine-readable `type`), collected with `new_block_args()`. A bare named character vector of descriptions, and the empty `character()`, are also accepted and normalized into this form, so existing registrations are unaffected. A single argument's fields are read back with `block_arg_description()`, `block_arg_example()` and `block_arg_type()`; block-level metadata (the whole argument set, worked examples, guidance and keywords) is tabulated across blocks via `block_metadata()`.

Usage

```
new_block_arg(description = NULL, example = NULL, type = NULL)

new_block_args(...)

block_arg_description(x, ...)

block_arg_example(x, ...)

block_arg_type(x, ...)

arg_string(description = NULL, required = TRUE)

arg_number(description = NULL, required = TRUE)

arg_integer(description = NULL, required = TRUE)

arg_boolean(description = NULL, required = TRUE)

arg_enum(values, description = NULL, required = TRUE)
```

```
arg_array(items, description = NULL, required = TRUE)

arg_object(..., description = NULL, required = TRUE)
```

Arguments

description	Human- and model-facing description of an argument value
example	A single worked value for an argument (or NULL)
type	Optional machine-readable type for the argument, built with the arg_*() descriptor constructors. A plain JSON-Schema-subset list; worked examples are validated against it
...	For new_block_args(), the per-argument block_arg objects (or bare description strings); for arg_object(), the named field descriptors; ignored by the block_arg_*() getters
x	A block_arg object, or a bare string taken as its description
required	Whether the field is required, when nested in an arg_object()
values	Allowed string values, for arg_enum()
items	Element descriptor, for arg_array()

Details

An argument's type is described with a small, dependency-free subset of JSON Schema, built with the arg_*() constructors: arg_string(), arg_number(), arg_integer() and arg_boolean() for scalars, arg_enum() for a fixed set of string values, arg_array() for a homogeneous list and arg_object() for a closed record of named fields (additionalProperties: false). Each returns a plain nested list mirroring the schema it denotes, consumed directly – blockr.ai binds it via ellmer::type_from_schema(), an MCP or raw tool schema reads the JSON as-is – and worked examples registered alongside an argument are validated against it (see register_block()). Semantic intent (e.g. "a column in the upstream data", "an R expression") is carried in description, not in the type vocabulary.

The complete worked configuration of a block is the assembly of its per-argument examples, keyed by argument name. When arguments interact, or several few-shot examples are wanted, complete configurations are instead supplied as a list via the examples argument of register_block() and supersede that assembly; combining multiple per-argument examples is intentionally not supported, as there is no safe way to form coherent whole-block configurations from them.

Value

new_block_arg() returns a block_arg and new_block_args() a block_args collection. The arg_*() constructors each return a plain JSON-Schema node (a list). The block_arg_*() getters return the corresponding field of a single argument (resolving a bare description string too).

Examples

```
new_block_args(
  n = new_block_arg(
```

```
    "Number of rows to return",
    example = 5L,
    type = arg_integer()
  )
)

arg_object(
  conditions = arg_array(
    arg_object(column = arg_string(), value = arg_string())
  ),
  operator = arg_enum(c("&", "|"))
)
```

---

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 = default_board_options(),
  ...,
  ctor = NULL,
  pkg = NULL,
  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
ctor, pkg	Constructor information (used for serialization)
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.

See Also

Real-world data blocks built on new_data_block() are provided by the blockr.io package, which sources data from a range of external formats.

---

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 = filebrowser_volumes(),
  ...
)

filebrowser_volumes(default = 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
default	Default volumes specification (use the blockr option "volumes" to override)

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.

See Also

The blockr.io package provides real-world blocks for sourcing external files.

`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 = "",
  ...,
  ctor = "new_link",
  pkg = pkg_name(),
  class = character()
)

update_link(x, delta)

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
ctor, pkg	Constructor information (used for serialization)
class	(Optional) link sub-class
x	Links object
delta	A named list of constructor argument values to apply on top of x's current fields.

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.

Partial-arg updates

update_link() is an S3 generic that produces a modified link by merging a named-list delta of constructor argument values onto an existing link. The default .link method passes the current as.list(x) (the link's from, to, input plus any extra *list-element* fields a sub-class adds) merged with the delta to the stored constructor. Sub-class extensions stored as attributes (rather than list elements) are **not** preserved through the default reconstruction — sub-class owners must register a class-specific method when their constructor uses attributes or a non-standard signature.

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.

See Also

The blockr.io package provides real-world blocks for parsing external data formats (e.g. `csv`, `xpt`).

`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.

See Also

Real-world plot blocks built on `new_plot_block()` are provided by the blockr.ggplot package.

`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 = abort_not_null, class = character())

is_plugin(x)

abort_not_null(x)

as_plugin(x)

plugin_server(x)

plugin_ui(x)

plugin_validator(x)

plugin_id(x)

board_plugins(x, ...)

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
...	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(new_board())

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 = default_stack_name,
  ...,
  ctor = "new_stack",
  pkg = pkg_name(),
  class = character()
)

default_stack_name()

is_stack(x)

stack_blocks(x)

stack_blocks(x) <- value

stack_name(x, name)

stack_name(x) <- value

update_stack(x, delta)

validate_stack(x)

as_stack(x)

stacks(...)

is_stacks(x)

as_stacks(x, ...)
```

Arguments

blocks	Set of blocks
name	Stack name
...	Extensibility
ctor, pkg	Constructor information (used for serialization)
class	(Optional) stack sub-class
x	Stack object
value	Replacement value
delta	A named list of constructor argument values to apply on top of x's current fields.

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.

Partial-arg updates

`update_stack()` is an S3 generic that produces a modified stack by merging a named-list `delta` of constructor argument values onto an existing stack. The default `.stack` method reconstructs the object via its stored constructor, treating the underlying character vector (block IDs) as `blocks` and every other attribute as a named constructor argument. Sub-class owners (e.g. `dock_stack` adding `color`) only need to register a method when their constructor deviates from this convention. The reserved delta key `blocks` replaces the member block IDs; every other key updates the named attribute.

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_text_block *Text block constructors*

Description

A text block produces (markdown styled) text, given some (optional) data input.

Usage

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

new_glue_block(text = 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_text_block() and new_block()
text	String evaluated using glue::glue()

Value

All blocks constructed via new_text_block() inherit from text_block.

Glue block

Using glue::glue(), this block allows evaluation of a text string in the context of datasets to produce (markdown formatted) text as block result.

---

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_fixed_block(expr, ...)

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()
expr	Quoted expression
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.

Fixed block

Mainly useful for testing and examples, this block applies a fixed transformation to its data input. No UI elements are exposed and the transformation consequently cannot be parametrized. The quoted expression passed as expr is expected to refer to the input data as data.

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", "left" 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()`.

See Also

Real-world transform blocks (e.g. `select`, `mutate`, `filter` and `join`) built on `new_transform_block()` are provided by the blockr.dplyr package.

`notify_user` *User notification plugin module*

Description

During the evaluation cycle of each block, conditions (errors, warnings and messages) may be raised. The default `notify_user` plugin surfaces these to the user as `shiny::showNotification()` toasts, displaying newly active conditions and clearing ones that are no longer active via `shiny::removeNotification()`. Each block's conditions (see `board_server()`) are tracked individually, so that a single block's change touches only its own notifications rather than re-processing the whole board.

Usage

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

notify_user_server(id, board, ...)

notify_user_ui(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()` and the UI component (e.g. `notify_user_ui()`) is expected to return shiny UI (i.e. `shiny::tagList()`). The server component (i.e. `notify_user_server()`) is called for the side effect of managing notifications and returns `NULL`.

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, ...)

restore_board(x, new, result, ..., session = get_session())

preserve_board_ui(id, board)

serialize_board(x, blocks, id = NULL, ..., session = get_session())
```

Arguments

`server, ui`	Server/UI for the plugin module
`id`	Namespace ID
`board`	The initial `board` object
`...`	Extra arguments passed from parent scope
`x`	The current `board` object
`new`	Serialized (list-based) representation of the new board
`result`	A `shiny::reactiveVal()` to hold the new board object
`session`	Shiny session
`blocks`	Block state reactive values

Details

The `session$reload()` that a restore triggers is handled by the app's `board_loader()` (passed to `serve()`), not by the plugin – the plugin server simply returns the `board` to restore and never reloads.

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 a `shiny::reactiveVal()` evaluating to `NULL` or the `board` to restore.

`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`. A `blockr_option()` `rand_id` can be set to swap out the function responsible for ID generation.

Usage

```
rand_names(
  old_names = character(0L),
  n = 1L,
  max_tries = 100L,
  id_fun = blockr_option("rand_id", NULL)
)

adjective_animal(n)

sample_letters(n)

resolve_ctor(ctor, ctor_pkg = NULL)

forward_ctor(x)

is_blockr_ctor(x)

ctor_name(x)

ctor_pkg(x)

ctor_fun(x)

to_sentence_case(x, replace = character(), with = character())

id_to_sentence_case(x)
```

Arguments

`old_names`	Disallowed IDs
`n`	Number of IDs to generate
`max_tries`	Max number of attempts to create IDs that do not intersect with `old_names`
`id_fun`	A function with a single argument `n` that generates random IDs. A value of `NULL` defaults to `ids::adjective_animal()` if available and `sample_letters` otherwise.

ctor	Function (either a string, a function or number used to index the call stack
ctor_pkg	The package where ctor is defined (either a string or NULL which will use the function environment)
x	Character vector to transform
replace, with	Mapped to base::gsub()

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()
rand_names(n = 5L)
rand_names(id_fun = sample_letters)
```

---

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 = NULL,
  icon = NULL,
  arguments = NULL,
  details = NULL,
  link = NULL,
  guidance = NULL,
  examples = list(),
  keywords = NULL,
```

```
  package = NULL,
  overwrite = FALSE
)

default_icon(category = default_category())

default_category()

suggested_categories()

list_blocks()

registry_id_from_block(block)

unregister_blocks(uid = list_blocks())

register_blocks(...)

available_blocks()

registry_metadata(blocks = list_blocks(), fields = "all")

create_block(id, ...)

register_package_blocks(which = "all", package = pkg_name(), overwrite = TRUE)
```

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.
icon	Icon
arguments	Block argument specification, either a new_block_args() object or a (possibly empty) named character vector of argument descriptions
details	Optional longer human-facing description (e.g. for a help popover), complementing the short description
link	Optional URL to a block's help or documentation page
guidance	Optional model-facing construction guidance (do/don't rules, enums, pitfalls), distinct from the human-facing details. Retrieved with block_metadata()
examples	Optional list of complete worked configurations (each a named list keyed by argument name); supersedes the per-argument example assembly. Retrieved with block_metadata()

keywords	Optional character vector of free-text search terms for block discovery, retrieved with block_metadata()
package	Package where constructor is defined (or NULL)
overwrite	Overwrite existing entry
block	Block object
...	For register_blocks(), arguments forwarded to register_block(); for new_block_args(), named new_block_arg() objects (or strings), one per constructor formal
blocks	Character vector of registry IDs
fields	Metadata fields
id	Block ID as reported by list_blocks()
which	Character vector of block constructor names to register (or "all"), keying into the YAML registry generated by the block_registration_roclet()

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) block_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 block_registry_entry object(s) invisibly, while unregister_blocks() returns NULL (invisibly). register_package_blocks() registers the blocks a package declares in its YAML registry (written by the block_registration_roclet()), likewise returning their entries invisibly. Listing via list_blocks() returns a character vector and a list of block_registry_entry object(s) for available_blocks(). create_block() returns a newly instantiated block object. A block's registered metadata is read with block_metadata() and the block_meta_*() accessors, and its argument specification is built and read with new_block_arg() and friends.

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 = blockr_app_plugins,
  options = blockr_app_options,
  loader = local_loader(),
  ...
)

blockr_app_plugins(x)

custom_plugins(x)

blockr_app_options(x)

custom_options(x)

blockr_app_ui(id, x, ...)

blockr_app_server(id, x, ...)

blockr_test_exports(x, rv, ...)
```

Arguments

x	Object
...	Generic consistency
id	Board namespace ID
data	Data inputs
plugins	Board plugins
options	Board options
loader	A board_loader() resolving the board to build for each request; defaults to local_loader(), the in-process save/restore handoff
rv	Board shiny::reactiveValues()

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 = get_session(), ...)

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

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

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

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

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

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

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

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.

`str_value` *Compact rendering*

Description

`str_value()` returns a compact string describing an object. It is the value-returning half of the compact rendering tier: where `utils::str()` only displays (it `cat()`s and returns `NULL`), `str_value()` returns the string, mirroring how `format()` returns what `print()` displays in the full, multi-line tier. The `utils::str()` methods are thin wrappers that display `str_value()`.

Usage

```
str_value(x, ...)

## Default S3 method:
str_value(x, ...)
```

Arguments

`x`	Object to render.
`...`	Generic consistency.

Details

A scalar object (a `block`, `stack`, `link`, `board_option` or `plugin`) renders as a single line; a container (`blocks`, `stacks`, `links`, `board_options`, `plugins`) and a whole `board` render as one element per line below a `<class[n]>` header. The `block` method lists a block's constructor inputs, marking the externally controllable ones (those reported by `external_ctrl_vars()`) with a trailing `*`; the `stack` method shows the stack name and its member block ids.

This is the blockr extension point for token-dense renderings such as a board summary. A home package surfaces a subclass's state by defining a `str_value()` method, typically extending the parent's via `NextMethod()` (the way `format.dock_stack()` appends a stack colour). `print()` and `format()` are unaffected and remain the full, multi-line tier.

Value

`str_value()` returns a length-one character vector (multi-line for containers). The `utils::str()` methods are called for their side effect (display) and return their `object` invisibly.

Examples

```
str_value(new_dataset_block())

str_value(new_stack(c("plot", "data"), name = "My stack"))

board <- new_board(c(a = new_dataset_block()))
str(board)
```

`trim_rv` *Remove entries from a* `reactiveValues` *object*

Description

shiny offers no public way to delete a key from a `shiny::reactiveValues()` object – assigning `NULL` stores a `NULL` value but leaves the key in `names()`. `trim_rv()` removes the named entries outright and invalidates the affected reactive dependencies, so a key can be truly dropped (and later re-added) – for instance when a variadic block argument is unlinked.

Usage

```
trim_rv(x, rm)
```

Arguments

`x`	A `reactiveValues` object.
`rm`	Character vector of keys to remove; all must be present in `x`.

Value

`x`, invisibly.

`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",
  envir = parent.frame(),
  asis = FALSE,
  use_glue = TRUE,
  pkg = pkg_name(envir)
)

log_fatal(..., envir = parent.frame())

log_error(..., envir = parent.frame())

log_warn(..., envir = parent.frame())

log_info(..., envir = parent.frame())

log_debug(..., envir = parent.frame())

log_trace(..., envir = parent.frame())

as_log_level(level)

fatal_log_level

error_log_level

warn_log_level

info_log_level

debug_log_level

trace_log_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"
`envir`	Environment where the logging call originated from

asis	Flag to disable re-wrapping of text to terminal width
use_glue	Flag to disable use of glue
pkg	Package name
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