Package: blockr.session (via r-universe)

November 19, 2026

Title Session Management for 'blockr'

Version 0.1.1

Description Persist, restore and manage 'blockr' boards from within a running app. Provides a 'manage_project' plugin for 'blockr.core' that replaces the built-in file upload and download interface with storage backed by the 'pins' package. On 'Posit Connect' each visitor reads and writes pins under their own account, with board sharing, visibility controls and version history.

License GPL (>= 3)

URL https://bristolmyerssquibb.github.io/blockr.session/, https://github.com/BristolMyersSquibb/blockr.session

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

Encoding UTF-8

Roxygen list(markdown = TRUE)

Imports blockr.core (>= 0.1.3), blockr.ui, shiny, pins (>= 1.4.0), jsonlite, glue, rlang, bsicons, htmltools, httr2, utils, zip

Suggests testthat (>= 3.0.0), blockr.dock (>= 0.1.3.9003), connectapi, knitr, rmarkdown, quarto, withr, xml2

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

VignetteBuilder quarto

Config/testthat/edition 3

Config/roxygen2/version 8.0.0

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

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

Date/Publication 2026-10-28 10:13:30 UTC

RemoteUrl https://github.com/BristolMyersSquibb/blockr.session

RemoteRef HEAD

RemoteSha 66d950549a905066a2860c1d39da1cf6cf320bc8

Contents

avatar_navbar_item 2
manage_project 3
rack-backend 4
rack_create 7
rack_load 8
rack_loader 9
user_pins_board 9

Index **11**

`avatar_navbar_item` *Navbar avatar*

Description

An item for the navbar of a blockr.dock board that shows the initials of the user signed in to the app, from `session$user`. A server that signs its visitors in, such as Posit Connect, sets it, and for a visitor who is not signed in the item draws nothing. Appended with `blockr.dock::custom_navbar()`, the avatar sits last on the bar.

Usage

```
avatar_navbar_item()
```

Value

A `blockr.dock::navbar_item()` with id `"avatar"`.

Examples

```
blockr.dock::is_navbar_item(avatar_navbar_item())

if (interactive()) {
  blockr.core::serve(
    blockr.dock::new_dock_board(),
    plugins = blockr.core::custom_plugins(manage_project()),
    navbar = blockr.dock::custom_navbar(avatar_navbar_item())
  )
}
```

`manage_project` *Project management*

Description

Enhanced session management with navbar-compatible UI. Provides a `blockr.core::preserve_board()` plugin with full navbar layout including workflows, version history, and editable title.

Usage

```
manage_project(server = manage_project_server, ui = manage_project_ui)

manage_project_server(id, board, ...)

manage_project_ui(id, x)
```

Arguments

`server`, `ui`	Server/UI for the plugin module
`id`	Namespace ID
`board`	Reactive values object containing board state
`...`	Extra arguments (may include dock object)
`x`	Board object

Value

See `blockr.core::preserve_board()`.

Crash recovery

Set the `session_autosave` `blockr.core::blockr_option()` to an interval in seconds to have unsaved work parked for recovery. On each tick the board is serialized and, when it differs from what was last parked, written to the configured backend as a *draft* (see `rack_create()`): an ordinary record at a reserved id, which overwrites in place rather than keeping a history and is held out of the workflow listing. An idle board costs one local serialization per tick and no backend traffic at all.

Recovery is what a draft is for, so nothing is published by it: a save remains the only way work reaches a record other people can load. A session holding a loaded record parks under that record, one slot reused; a board with no record yet parks under an opaque per-session id. Saving clears the latter. A later session lists what is left and offers each entry for restore or discard, leaving anything untouched available next time; the offer is reached from the navbar or by loading the app with a bare `recover` query parameter. A draft against a saved workflow is dropped by the save that supersedes it and otherwise kept, there being one per workflow however long anyone edits. A draft of a board with no record yet does accumulate, so it is swept once it passes the `session_draft_ttl` option's retention, in days, seven by default – measured from when the draft was opened, so set it well clear of how long a session typically runs.

Examples

```
plg <- manage_project()
blockr.core::is_plugin(plg)
```

---

rack-backend *Rack storage backend contract*

Description

The rack layer is the storage abstraction behind blockr session persistence: it turns a save into a stored, versioned record and a load back into board state. The shipped backend dispatches on **pins** boards (see `user_pins_board()`), but the contract is storage-agnostic – a database or document store is supported by implementing the generics documented here and carrying the `rack_backend` class, which is what wiring a non-pins backend into the `session_mgmt_backend` `blockr.core::blockr_option()` requires. The high-level `rack_create()`, `rack_append()` and `rack_load()` entry points that application code calls are built on these generics.

Usage

```
new_rack_id(id, version = NULL, user = NULL, ..., class = character())

new_rack_record(id, name, ..., class = character())

rack_content_hash(id, backend, ...)

rack_name(id, backend, ...)

rack_rename(id, backend, name, ...)

rack_exists(id, backend, ...)

rack_list(backend, tags = NULL, ...)

as_rack_id(x, backend, ...)

rack_info(id, backend, ...)

rack_download(id, backend, ...)

rack_upload(backend, path, ...)

rack_delete(id, backend, ...)

rack_purge(id, backend, ...)

rack_capabilities(backend, ...)
```

```
rack_tags(id, backend, ...)

rack_set_tags(id, backend, tags, ...)

rack_acl(id, backend, ...)

rack_set_acl(id, backend, acl_type, ...)

rack_share(id, backend, with_sub, ...)

rack_unshare(id, backend, with_sub, ...)

rack_shares(id, backend, ...)

rack_find_users(backend, query, ...)
```

Arguments

id	A `rack_id` identifying a stored record, for the accessor and mutator generics. For `new_rack_id()` and `new_rack_record()`, the record's storage id: a non-empty string, stable and independent of the display name.
version	For `new_rack_id()`, an optional version string pinning the `rack_id` to one stored version; `NULL` tracks the latest.
user	For `new_rack_id()`, the owning account, used to reach records stored under another user's namespace.
...	Passed on to methods; for the constructors, extra named fields stored on the object.
class	Character vector prepended to the object's class, so a backend carries its own `rack_id` / `rack_record` subclass.
name	Display name for the record, written to the backend's native name field and never used as the storage key; `NULL` keeps the current name.
backend	A rack backend: the storage object dispatched on. A **pins** board is recognized directly; any other backend carries the `rack_backend` class so the `session_mgmt_backend` option accepts it.
tags	Character vector of tags: the filter applied by `rack_list()` (`NULL` for none), or the tag set written by `rack_set_tags()`.
x	The object to coerce with `as_rack_id()`: an existing `rack_id` (returned unchanged), or a component bag – a list of identifier fields (`id`, and optionally `version` and `user`) or a `rack_record` – turned into the backend's own `rack_id` subclass.
path	Path to the local file to upload as a new version.
acl_type	Access-control level to set, backend-defined (e.g. `"private"`, `"acl"`, `"all"`).
with_sub	Backend identifier of the principal to share a record with or revoke.
query	Search prefix for `rack_find_users()`.

Details

Two kinds of object flow through the contract. A *backend* is the board or store itself, the dispatch target for `as_rack_id()`, `rack_capabilities()`, `rack_list()`, `rack_upload()` and `rack_find_users()`. A `rack_id` is a handle to one stored record, the dispatch target for the remaining generics. `as_rack_id()` coerces a raw identifier – a component list or a `rack_record` listing row – into the backend's own `rack_id` subclass (via `new_rack_id(..., class = )`) from the components (`id`, `version`, `user`) carried in a session URL, and returns an existing `rack_id` unchanged. `rack_list()` returns `new_rack_record()` summaries, the listing rows the workflow menu renders.

Value

`new_rack_id()` and `new_rack_record()` return a `rack_id` and `rack_record` respectively. `rack_capabilities()` returns the named list of flags above. `rack_list()` returns a list of `rack_records` and `rack_info()` a data frame of versions (columns `version`, `created`, `ref`, newest first). `as_rack_id()`, `rack_upload()` and `rack_rename()` return a `rack_id`; `rack_download()` a local file path; `rack_name()` a string; `rack_exists()` a scalar logical; `rack_content_hash()` the stored payload hash; `rack_tags()`, `rack_acl()` and `rack_shares()` the stored tags, access level and shares. The mutators (`rack_set_tags()`, `rack_set_acl()`, `rack_share()`, `rack_unshare()`, `rack_delete()`, `rack_purge()`) return invisibly.

Capabilities

`rack_capabilities()` is the feature switch: it returns a named list of logical flags declaring which optional features the backend supports, and the management UI gates the matching generics on it.

- `versioning`: a version history is kept per record: `rack_info()` lists it and `rack_download()` / `rack_delete()` accept a version.
- `tags`: `rack_tags()` and `rack_set_tags()` are implemented.
- `metadata`: a display name (`rack_name()`, `rack_rename()`) and a content hash (`rack_content_hash()`) are stored alongside the payload.
- `sharing`: `rack_share()`, `rack_unshare()` and `rack_shares()` are implemented.
- `visibility`: `rack_acl()` and `rack_set_acl()` are implemented.
- `user_discovery`: `rack_find_users()` is implemented.

A minimal backend implements only the core create / load / list set – `as_rack_id()`, `rack_upload()`, `rack_download()`, `rack_exists()`, `rack_list()`, `rack_info()` and `rack_name()` – and returns `FALSE` for every optional capability; the generics behind a `FALSE` flag are then never reached and need no method.

See Also

`rack_create()`, `rack_append()` and `rack_load()` for the high-level save / load API built on the contract, and `user_pins_board()` for the default **pins** backend.

rack_create *Create or append to a session record on a rack backend*

Description

`rack_create()` serialises `data` to JSON and stores it as a **new** record keyed on `id` – the board's own stable id, so the record id and the board id match. It is a strict insert: it errors (class `rack_create_exists`) if `id` already names a record rather than appending a version. `name` is written to the backend's native display field. `rack_append()` adds a **new version** to the existing record identified by `id`, erroring (class `rack_append_missing`) if there is none, and never touches the name. Together they replace the former `rack_save()`, separating insert from append. To change a record's name, use `rack_rename()`. `rack_records()` lists what a backend holds, of one kind at a time.

Usage

```
rack_create(backend, data, id, name, draft = FALSE, ...)

rack_append(id, backend, data, ...)

rack_records(backend, draft = FALSE, ...)
```

Arguments

`backend`	A rack backend object (e.g. a `pins_board`).
`data`	An R object to serialise and store (typically the session list returned by the blockr session machinery).
`id`	For `rack_create()`, the storage id to key the new record on (typically the board id); errors if it already names a record. For `rack_append()`, the `rack_id` of the record to add a version to.
`name`	Character scalar. The display name for the new record.
`draft`	Which records this concerns: `FALSE` for one the user saved, or a draft kind, `"session"` or `"record"`. A `rack_records()` call also accepts `TRUE`, matching a draft of either kind.
`...`	Additional arguments forwarded to `rack_upload()` or, for `rack_records()`, to `rack_list()`.

Value

`rack_create()` and `rack_append()` return a `rack_id` object identifying the newly created version, `rack_records()` a list of `rack_records`.

Drafts

A *draft* is unsaved work parked for crash recovery. It is an ordinary rack record in every respect a backend can see – same store, same tags, read with `rack_load()` and dropped with `rack_purge()` – and all that sets it apart is a reserved id, minted by `rack_create(draft = )` and matched by

`rack_records(draft = )`. Nothing below this API knows drafts exist, so a backend needs no draft support of its own.

Passing `draft` also relaxes what `rack_create()` does, because a draft is a single slot rather than a history: the id-already-taken check is skipped, so a repeated write overwrites, and the write is unversioned. With `draft = FALSE` the reserved namespace is refused instead, so a user cannot save a record into it.

A draft is of one *kind*. A `record` draft follows a saved workflow and is keyed on it, so its id is the same in every session: there is one slot per workflow by construction, it is overwritten as the work changes, and the save it was shadowing drops it. Nothing about it expires, since a workflow bounds it. A `session` draft belongs to a board with no record yet, keyed on an opaque per-session id, and those do accumulate – so one carries the time it was opened and is swept on a retention read straight off the id. That needs neither the optional `metadata` capability nor any field beyond those `new_rack_record()` guarantees. The key is hashed into the id rather than embedded, since a workflow id may hold characters a backend rejects in a record name.

The id a draft write returns carries no version: a draft keeps one, and the next write to that slot deletes it.

See Also

`rack_load()` for the complementary load function, `rack_rename()` to change a record's name, `rack_upload()` and `rack_list()` for the underlying generics.

---

rack_load *Load a session from a rack backend*

Description

Downloads and parses a stored session identified by `id` from the given `backend`. This is a convenience wrapper around `rack_download()` that additionally deserialises the JSON payload into an R object.

Usage

```
rack_load(id, backend, ...)
```

Arguments

`id`	A `rack_id` object identifying the session to load.
`backend`	A rack backend object (e.g. a `pins_board`).
`...`	Additional arguments forwarded to `rack_download()`.

Value

The deserialised session data as an R object (typically a named list).

See Also

rack_create() and rack_append() for the complementary save functions, rack_download() for the underlying download generic.

---

rack_loader *Rack-backed board loader*

Description

A blockr.core::board_loader() for blockr.core::serve() that picks the board to build from the request URL. The board named by the URL handle (board_name / user / version) loads from the session_mgmt_backend backend; a new handle (minted when the user hits New) yields an empty board under that fresh id, so saving it forks a distinct record rather than overwriting the served board; a request without any handle (a cold load) gets a cleared copy of the served board. The backend is resolved with the loader's own request (at the GET, before any session) or session (at the WS connect), so a user-scoped backend such as user_pins_board() resolves under the *visitor's* own Posit Connect credentials at both phases; a board the visitor may not read does not resolve, whatever the user / board_name in the URL. Pair it with manage_project() when calling blockr.core::serve():

```
serve(
  board,
  plugins = custom_plugins(manage_project()),
  loader = rack_loader()
)
```

Usage

```
rack_loader()
```

Value

A blockr.core::board_loader() object.

---

user_pins_board *Posit Connect storage backend with graceful fallback*

Description

Resolves the storage backend for the session_mgmt_backend blockr.core::blockr_option() – of which it is the default – in three tiers, degrading instead of erroring on a missing credential:

Usage

```
user_pins_board(session = get_session())
```

Arguments

session	Shiny session whose request carries the Connect user session token; defaults to the current reactive domain.

Details

1. When the visitor carries a Posit Connect user session token (the Connect API Integration is enabled), it is exchanged via `connectapi::connect()` for a viewer-scoped API key, so each user reads and writes pins under their own namespace (`pins::board_connect()`). Requires the **connectapi** package.
2. With no visitor token but application Connect credentials in the environment (`CONNECT_SERVER` and `CONNECT_API_KEY`), a board on the application's own account.
3. Otherwise (e.g. local development, off Connect), `pins::board_local()`.

Set the `session_connect_tag` `blockr.core::blockr_option()` to an existing Connect tag's name (or a `"Category/Name"` path) to have the workflow listing filter server-side by that native tag; saves then apply it. A top-level tag is a category: naming one lists everything tagged beneath it, but saves cannot apply it, as Connect assigns only nested tags. Unset, the listing returns every pin and checks blockr membership only on load.

Value

A `pins_board`: a viewer- or application-scoped `board_connect`, or `board_local` when no Connect credentials are available.

Examples

```
## Not run:
  # Resolves the backend for the current visitor; call from within a
  # running Shiny app, as it reads the session request for a Connect
  # user session token. It is the default value of the
  # `session_mgmt_backend` blockr option.
  board <- user_pins_board()

## End(Not run)
```

Index

as_rack_id *(rack-backend)*, 4
as_rack_id(), *6*
avatar_navbar_item, 2

blockr.core::blockr_option(), *3, 4*, *9, 10*
blockr.core::board_loader(), *9*
blockr.core::preserve_board(), *3*
blockr.core::serve(), *9*
blockr.dock::custom_navbar(), *2*
blockr.dock::navbar_item(), *2*

manage_project, 3
manage_project(), *9*
manage_project_server *(manage_project)*, 3
manage_project_ui *(manage_project)*, 3

new_rack_id *(rack-backend)*, 4
new_rack_record *(rack-backend)*, 4
new_rack_record(), *6*

pins::board_connect(), *10*
pins::board_local(), *10*

rack-backend, 4
rack_acl *(rack-backend)*, 4
rack_append *(rack_create)*, 7
rack_append(), *4*, *6*, *9*
rack_capabilities *(rack-backend)*, 4
rack_content_hash *(rack-backend)*, 4
rack_create, 7
rack_create(), *3, 4*, *6*, *9*
rack_delete *(rack-backend)*, 4
rack_download *(rack-backend)*, 4
rack_download(), *8, 9*
rack_exists *(rack-backend)*, 4
rack_find_users *(rack-backend)*, 4
rack_info *(rack-backend)*, 4
rack_list *(rack-backend)*, 4
rack_list(), *7, 8*
rack_load, 8
rack_load(), *4*, *6–8*
rack_loader, 9
rack_name *(rack-backend)*, 4
rack_purge *(rack-backend)*, 4
rack_records *(rack_create)*, 7
rack_rename *(rack-backend)*, 4
rack_set_acl *(rack-backend)*, 4
rack_set_tags *(rack-backend)*, 4
rack_share *(rack-backend)*, 4
rack_shares *(rack-backend)*, 4
rack_tags *(rack-backend)*, 4
rack_unshare *(rack-backend)*, 4
rack_upload *(rack-backend)*, 4
rack_upload(), *7, 8*

user_pins_board, 9
user_pins_board(), *4*, *6*, *9*