

Package: ember (via r-universe)

October 10, 2026

Title Reactive Notebooks for R

Version 0.0.0.9000

Description A reactive notebook engine for R in the style of Pluto.jl.
Cells are read statically into a dependency graph, so editing a
cell reruns the cells that depend on it.

License MIT + file LICENSE

Copyright See inst/COPYRIGHTS.

Encoding UTF-8

Depends R (>= 4.1.0)

Imports commonmark, httpuv, jsonlite, later, processx, RcppMsgPack,
renv, stats, utils

Suggests htmltools, knitr, testthat (>= 3.0.0), withr

Config/testthat/edition 3

Config/pak/sysreqs make zlib1g-dev

Repository <https://jowch.r-universe.dev>

Date/Publication 2026-10-10 18:25:18 UTC

RemoteUrl <https://github.com/jowch/Ember>

RemoteRef HEAD

RemoteSha 81320c5461bb4ea3c14f8c869b2dd24d2b1886fb

Contents

edit_notebook	2
graph_queries	3
notebook_graph	5
notebooks	6
packages	9
read_cell	12
server	13

Index	16
--------------	-----------

edit_notebook	<i>Stage edits to a notebook</i>
---------------	----------------------------------

Description

`edit_notebook()` applies a batch of edit ops to a notebook at once. Edits are staged: they change the code, the graph and the file, but never run anything. `set_code()`, `insert_cell()`, `delete_cell()`, `move_cell()`, `fold_cell()` and `disable_cell()` build the ops; they are plain data until passed to `edit_notebook()`.

Usage

```
edit_notebook(nb, ...)

set_code(cell, code, expected = NULL)

insert_cell(index, code = "", id = NULL)

delete_cell(cell)

move_cell(cell, index)

fold_cell(cell, folded = TRUE)

disable_cell(cell, disabled = TRUE)
```

Arguments

<code>nb</code>	An <code>ember_notebook</code> handle.
<code>...</code>	One or more ops made by <code>set_code()</code> , <code>insert_cell()</code> , <code>delete_cell()</code> , <code>move_cell()</code> , <code>fold_cell()</code> , <code>disable_cell()</code> , applied in order. Indexes are display positions after the previous ops in the same call. An op may name, by id, a cell an earlier <code>insert_cell()</code> in the same call adds.
<code>cell</code>	A cell id or 1-based display index.
<code>code</code>	The cell's new code (a single string).
<code>expected</code>	The code the caller believes the cell currently has, as a check before applying (Endeavor's optimistic-concurrency check). <code>NULL</code> skips the check.
<code>index</code>	1-based display position: where to insert (<code>insert_cell()</code>) or move to (<code>move_cell()</code>).
<code>id</code>	The new cell's id (a 36-character UUID), for a caller that names the cell itself, as the browser does. <code>NULL</code> generates one. A cell of only <code>#</code> lines is text; there is no separate argument for its kind.
<code>folded</code>	<code>TRUE</code> to fold the cell's code away in a view, <code>FALSE</code> to show it.
<code>disabled</code>	<code>TRUE</code> to disable the cell, <code>FALSE</code> to re-enable it. A disabled cell and every cell that depends on it stop running, keep their last output dimmed, and lose their globals, as if deleted; text cells can't be disabled.

Value

`edit_notebook()` returns, invisibly, a list with `inserted` (ids of inserted cells, in op order) and `seq` (the state's seq after the batch). If any op in the batch cannot apply (an unknown id, a mismatched expected, an attempt to disable a text cell, or code containing a line Ember reserves as a cell or footer marker), the whole batch is refused and an `ember_refused` condition is signalled; none of the ops take effect.

`set_code()`, `insert_cell()`, `delete_cell()`, `move_cell()`, `fold_cell()` and `disable_cell()` each return an `ember_op`, a plain list describing the edit; they do nothing on their own.

Examples

```
## Not run:
nb <- open_notebook("growth.R")
edit_notebook(nb, set_code(2, "deg <- 3"))
edit_notebook(nb, insert_cell(3, "summary(fit)"), fold_cell(1))
edit_notebook(nb, disable_cell(2))

## End(Not run)
```

graph_queries

Read-only queries over a notebook graph

Description

Queries over an `ember_graph` (see [notebook_graph](#)): neighbours, run order, per-cell summaries, which cells an edit or a worker report affects, and what changed between two graph builds.

Usage

```
upstream(graph, id, transitive = FALSE)

downstream(graph, id, transitive = FALSE)

run_order(graph, ids = NULL)

affected(old, new, changed)

cell_errors(graph, id)

blocked_cells(graph)

cell_summary(graph, id)

changed_cells(old, new)

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

```
## S3 method for class 'ember_graph_error'
format(x, ...)
```

Arguments

graph	An ember_graph.
id	A cell id.
transitive	FALSE (the default) returns direct neighbours only; TRUE returns every ancestor (upstream()) or descendant (downstream()), in run order.
ids	Character vector of cell ids, or NULL for the whole notebook's run order.
old, new	Two ember_graph values built for the same notebook, old before an edit and new after.
changed	Character vector of ids that were just edited.
x	An ember_graph, for print(), or an ember_graph_error, for format().
...	Unused; present for consistency with the generics.

Value

upstream(), downstream() return a character vector of cell ids.

run_order() returns graph\$order filtered to ids (or all of it when ids is NULL), in run order.

affected() returns, in run order, the cells to rerun or mark stale after editing changed: changed itself, plus everything downstream of it in new, plus anything downstream of it in old that still exists in new.

cell_errors() returns a list of ember_graph_error reported on id.

blocked_cells() returns a character vector of ids that can't run (named in any graph error).

cell_summary() returns a list for one cell: list(definitions, learned, references, packages, attaches, upstream, downstream, errors, notes, formulas). definitions is static definitions only (learned ones subtracted out, since learned reports them separately).

changed_cells() returns a character vector of ids whose resolved view (definitions, references, packages, upstream set, or errors) differs between old and new, including ids added or deleted; present ids come first in new's run order, followed by deleted ids in old's run order.

print.ember_graph() prints one line per cell (id, definitions, upstream ids) followed by its errors, and returns x invisibly.

format.ember_graph_error() returns a one-line string, "[<kind>] <message>".

Examples

```
g <- notebook_graph(
  cells = c(a = "x <- 1", b = "y <- x + 1")
)
upstream(g, "b")
downstream(g, "a", transitive = TRUE)
run_order(g)
cell_summary(g, "b")$references
blocked_cells(g)
```

notebook_graph

*Build and update the dependency graph***Description**

`notebook_graph()` builds the whole-notebook dependency graph from the cells' code, the installed packages' exports, the disabled cells, and whatever the worker has learned at run time. It is pure and deterministic: the same inputs always give an identical graph. `graph_learn()` folds a new fact reported by the worker (names it created, formula columns it read, or global settings it changed) into a graph, rebuilding only what that fact can change.

Usage

```
notebook_graph(cells, exports = list(),
  learned = NULL, disabled = character(), previous = NULL, read_file = NULL)

graph_learn(graph, cell, definitions = NULL, references = NULL,
  settings = NULL)
```

Arguments

<code>cells</code>	Named character vector, cell id -> code, in display order. Markdown cells are passed as empty strings (or left out); they take part in the order and nothing else.
<code>exports</code>	Named list, package name -> character vector of exported names (as from <code>getNamespaceExports()</code>), for the packages that are installed. A package not in the list contributes no edges.
<code>learned</code>	<code>list(definitions = <id -> character>, references = <id -> character>, settings = <id -> character>)</code> from the footer, from formula checks and from the settings the worker saw cells change (as keys such as "option:digits", "env:TZ", "wd"); NULL means none. Ids not in cells are dropped.
<code>disabled</code>	Character ids of disabled cells (see <code>disable_cell()</code>). Ids not in cells are dropped. Disabled cells are analysed like any other, but satisfy no other cell's reference and count towards no error rule but parse.
<code>previous</code>	An earlier <code>ember_graph</code> for the same notebook, or NULL. Analyses are reused for cells whose code is identical and whose sourced files still read the same; everything else is recomputed. Passing <code>previous</code> never changes the result, only the work done.
<code>read_file</code>	Reader for <code>source()</code> paths; see read_cell .
<code>graph</code>	An <code>ember_graph</code> , as returned by <code>notebook_graph()</code> or <code>graph_learn()</code> .
<code>cell</code>	The id of the cell the worker ran.
<code>definitions</code>	Character vector of names the cell defined at run time (replaces, not adds to, any previously learned definitions for this cell), or NULL to leave them unchanged.
<code>references</code>	Character vector of names the cell's formulas or dynamic code read at run time, or NULL to leave them unchanged.
<code>settings</code>	Character vector of setting keys the cell changed at run time (replaces any previously learned settings for this cell; <code>character()</code> drops them), or NULL to leave them unchanged.

Details

A settings cell (one that changes a global setting, statically or as learned) runs first, after the cells it reads, and every later cell depends on it through a "setting" edge. A setting or an attached package in two cells is a graph error (`setting_conflict`, `package_conflict`).

`graph_learn()` rebuilds with `previous = graph`, so only the affected cell's analysis changes and the rest of the graph is reused. Learned definitions take part in the multiple-definitions rule and in edges exactly like statically read ones; learned references go through the same edge resolution as static references.

Value

Both return an `ember_graph`: a list with `ids`, `settings` (the settings cells, in display order), `analyses`, `learned`, `exports`, `cells` (per-cell resolved view), `edges` (a data frame of `from`, `to`, `name`, `via`), `upstream`, `downstream`, `order` (every id in run order), `errors` (a list of `ember_graph_error`), and `reread` (ids whose analysis was actually recomputed in this build).

notebooks

Open, run and inspect a notebook

Description

The session API: open or create a notebook file, stage runs, and read back its state. A notebook handle (`ember_notebook`) is an opaque environment owned by a background shell; callers never touch its fields directly, they read it through `notebook_snapshot()`, `dependency_graph()` and `notebook_state()`.

Usage

```
open_notebook(path, repos = ember_repos(), cache = cache_dir())
```

```
new_notebook(path, repos = ember_repos(), cache = cache_dir())
```

```
close_notebook(nb)
```

```
move_notebook(nb, path)
```

```
set_cell_change_mode(nb, on_cell_change = c("autorun", "lazy"))
```

```
allow_execution(nb)
```

```
run_cells(nb, ids = NULL, wait = FALSE, timeout = 60)
```

```
interrupt_notebook(nb)
```

```
restart_notebook(nb)
```

```

notebook_snapshot(nb)

dependency_graph(nb)

notebook_state(nb)

render_png(nb, cell, width = NULL, height = NULL, res = 96, timeout = 5)

on_notebook_event(nb, fn)

wait_for(nb, predicate = is_idle, timeout = 10)

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

```

Arguments

path	Path to the notebook's .R file. For <code>open_notebook()</code> , an existing file (one without Ember's header opens too). For <code>new_notebook()</code> , a path that does not yet exist. For <code>move_notebook()</code> , the new path.
repos	<code>ember_repos()</code> : where indexes and packages come from. See packages .
cache	<code>cache_dir()</code> : where libraries, indexes and renv's cache live. See packages .
nb	An <code>ember_notebook</code> handle, as returned by <code>open_notebook()</code> or <code>new_notebook()</code> .
on_cell_change	"autorun" (edit-then-run reruns dependents automatically) or "lazy" (dependents are marked stale but don't rerun until asked). Stored in the file's header.
ids	Cell ids or 1-based display indexes to run, or NULL to run every code cell.
wait	TRUE blocks until the run queue is empty or timeout passes, servicing pending work meanwhile. Only for the console and tests; code already running inside an event callback must use <code>wait = FALSE</code> and <code>on_notebook_event()</code> instead.
timeout	Seconds to wait before giving up (<code>run_cells()</code> , <code>render_png()</code>) or before <code>wait_for()</code> returns FALSE.
cell	A cell id or 1-based display index.
width,height	Requested pixel size for <code>render_png()</code> . NULL for both returns the output's stored image unchanged.
res	Pixel density (dots per inch) for <code>render_png()</code> 's re-render, e.g. <code>96 * devicePixelRatio</code> . Ignored when width and height are both NULL.
fn	Function called as <code>fn(note)</code> for every notification; see Details.
predicate	Function of a snapshot (as returned by <code>notebook_snapshot()</code>) used by <code>wait_for()</code> ; it returns TRUE once the wait is satisfied. Defaults to "nothing queued or running".
x	An <code>ember_notebook</code> , for <code>print()</code> .
...	Unused; present for consistency with <code>print()</code> generics.

Details

Opening a notebook (`open_notebook()`) only parses the file and builds its graph; no worker starts and nothing runs until `run_cells()` or `allow_execution()` is called. `run_cells()` runs the requested cells and any unrun, stale or failed ancestors first, in dependency order, one at a time. A cell with a graph error of its own, or a text cell passed by id, is returned in `skipped` rather than `queued`; a dependent of a failed or graph-broken cell still runs, and fails on its own if it needs what that cell would have provided.

The worker always starts from the notebook's own package library: its lock is resolved and installed as needed (see [packages](#)), so there is no separate `library` argument to hand it a different one.

`on_notebook_event()`'s `note` is `list(kind, seq, ...)` with `kind` one of "notebook_opened", "cell_state", "topology_changed", "file_saved", "execution_done", "notebook_shut_down".

Value

`open_notebook()` and `new_notebook()` return an `ember_notebook` handle.

`close_notebook()` returns `TRUE` if the notebook was still in safe preview (no worker running).

`move_notebook()` returns the new path. A running R process's working directory follows the file, unless the notebook changed it.

`allow_execution()` returns `TRUE` if execution was already allowed.

`run_cells()` returns, invisibly, `list(accepted, queued, skipped, finished, timed_out)`.

`notebook_snapshot()` returns a plain list describing the notebook's current, safe-to-keep state: `list(id, path, seq, read_only, allowed, process, restart_offered, worker_message, worker_memory, problems, order, cells, packages)`, where `process` is one of "preview", "starting", "ready", "busy", "stopped", each cell's `variables` is a list of `list(name, type)` for the globals it owns (no value or kind: a value the notebook never printed stays out of this API), `worker_memory` is the worker's last-reported memory in bytes (NULL until sampled for the current worker), and `packages` is `package_status(nb)`'s value (see [packages](#)).

`dependency_graph()` returns the notebook's current `ember_graph` (see [notebook_graph](#)), reflecting staged edits.

`notebook_state()` returns the session's immutable internal state, for advanced callers (the step-4 UI).

`render_png()` returns `list(png = <raw> | NULL, mime = <mime type>)`.

`on_notebook_event()` returns a function that unsubscribes when called.

`wait_for()` returns `TRUE` if the predicate held, `FALSE` on timeout.

`set_cell_change_mode()`, `interrupt_notebook()` return NULL, invisibly. `restart_notebook()` returns the dispatch reply, invisibly.

Examples

```
## Not run:
nb <- open_notebook("growth.R")
nb
run_cells(nb, 4, wait = TRUE)
snap <- notebook_snapshot(nb)
snap$cells[[4]]$output$text
```

```

interrupt_notebook(nb)
restart_notebook(nb)
close_notebook(nb)

## End(Not run)

```

packages	<i>A notebook's packages: status, the date move and running outside the engine</i>
----------	--

Description

A notebook detects the packages its code uses, resolves and locks their versions at its snapshot date, and installs a library with renv. Nothing about packages appears in the ordinary calls (`edit_notebook()`, `run_cells()`): the lock fills in as soon as an edit is applied, and the install starts on the first run. These functions are for reading that status, changing the package set directly, moving the snapshot date, and running or cleaning up outside a session.

Usage

```

package_status(nb)

add_extra_package(name)

remove_extra_package(name)

preview_date(nb, date = Sys.Date(), wait = TRUE, apply = FALSE, timeout = 60)

set_date(nb, date)

run(path, repos = ember_repos(), cache = cache_dir(), echo = TRUE)

clean(max_age = 60, cache = TRUE, dry_run = FALSE, dir = cache_dir(),
      now = Sys.time())

ember_repos(cran = "https://packagemanager.posit.co/cran",
            bioc = "https://packagemanager.posit.co/bioconductor",
            bioc_config = "https://bioconductor.org/config.yaml")

cache_dir()

```

Arguments

nb	An <code>ember_notebook</code> handle.
name	A package name, for <code>add_extra_package()/remove_extra_package()</code> .
date	A date (anything <code>as.Date()</code> accepts). <code>preview_date(nb, Sys.Date())</code> is "update all".

<code>wait</code>	TRUE blocks until the preview is ready (or failed) or timeout passes, servicing pending work meanwhile, as <code>run_cells()</code> 's <code>wait</code> does. Only for the console and tests; code running inside an event callback must use <code>wait = FALSE</code> and <code>on_notebook_event()</code> instead.
<code>apply</code>	TRUE applies the preview itself once it is ready, the same change <code>set_date()</code> would make, unless doing so would restart a package the worker has already loaded; the Packages tab's Update button uses this. FALSE (the default) only ever previews: applying is always a separate, explicit <code>set_date()</code> call.
<code>timeout</code>	Seconds to wait before giving up.
<code>path</code>	Path to a notebook's .R file, for <code>run()</code> .
<code>repos</code>	An <code>ember_repos()</code> value: where indexes and packages come from. Passed to <code>open_notebook()/new_notebook()</code> and to <code>run()</code> .
<code>cache</code>	A <code>cache_dir()</code> -shaped path: where libraries, indexes and renv's cache live. Passed to <code>open_notebook()/new_notebook()</code> , <code>run()</code> and <code>clean()</code> .
<code>echo</code>	TRUE to stream the install's and the file's own output to the console, for <code>run()</code> .
<code>max_age</code>	Days since last use after which an unused library or a cached index is deleted by <code>clean()</code> .
<code>dry_run</code>	TRUE lists what <code>clean()</code> would delete without deleting anything.
<code>dir</code>	The cache directory <code>clean()</code> operates on.
<code>now</code>	The time <code>clean()</code> measures ages against; a parameter mainly for tests.
<code>cran, bioc</code>	Base repository URLs (no date), for <code>ember_repos()</code> . The defaults are the public Posit Package Manager; an institution's own Package Manager, or a <code>file://</code> folder laid out the same way, can replace them.
<code>bioc_config</code>	Where Bioconductor's list of releases comes from, for <code>ember_repos()</code> : <code>bioconductor.org</code> 's <code>config.yaml</code> , or a <code>file://</code> copy. Fetched when a notebook first needs Bioconductor and kept in cache for a day; the last copy fetched is used when a fetch fails. NA turns the lookup off, and with it Bioconductor.

Details

`set_date()` changes the header's snapshot and the lock in one step and saves the file; the library installs once execution is allowed, and if a package the worker has already loaded changed version, the worker restarts once the new library is ready, leaving every cell not run. `set_date()` and an `apply = TRUE` preview that needs no restart both apply the proposal the same way internally: neither is allowed to apply anything but the current, ready proposal.

A page connected to the notebook sees an `apply = TRUE` preview as `ember$packages$update: list(date, status, restart, changes, message)`, present only while such a preview exists. A plain `preview_date()` call (no `apply`) is never shown there: it is this function's own, R-side preview, not the Packages tab's.

`run()` does not resolve: the file's lock is the authority. If the code names a package the lock lacks, `run()` says so and runs anyway (R reports the missing package), since resolving would change the file. Building the library blocks the caller, the same installer and layout a session uses.

`clean()` never deletes a library an open session in this process is using, and never touches a user's own renv projects: Ember keeps its own renv cache under `cache_dir()`.

Value

`package_status()` returns a plain list, the same value as `notebook_snapshot(nb)$packages`: `list(snapshot, r_version, bioc_version, library = list(status, path, progress, message, log, failures), packages = data.frame(name, version, source, direct, status, message), problems, proposal, plan)`. Per package, status is one of "installed", "installing", "missing", "failed" (named by `library$failures`, or a direct package that needs one of them), "not_installed" (every other package in a library that failed: its staging folder is never renamed into place, so nothing in that install actually landed), "not_found". plan is NULL or `list(install = <n packages>, restart = <names>)`, what running would do (a "not_installed" package counts toward it exactly as "missing" does).

`library$log` is the installer's last 200 lines from the most recent failed install, whatever status is now: it isn't cleared by a later successful install, so it can be stale (describing an earlier failure) once status has moved past "failed". `character()` before any install has ever failed. `library$failures` is a data frame package, kind ("compile", "configure", "dependency", "download", "other"), detail (the missing dependency's name for "dependency", the first ERROR line for "other", NA otherwise), needed_by (a list column: the direct packages whose dependency closure reaches this one). Empty (0 rows) unless status is "failed".

`add_extra_package()/remove_extra_package()` return an `ember_op`, a plain list, for use with `edit_notebook()`; they do nothing on their own. `remove_extra_package()` is refused through `edit_notebook()` when the name is also named directly in code (removing it from the header would do nothing): edit the code instead.

`preview_date()` returns `list(date, status, changes, problems, restart)`, where status is "fetching", "ready" or "failed", and changes is NULL or a data frame name, from, to, change ("added", "removed", "upgraded", "downgraded", "source"). restart is `character()` unless `apply = TRUE` left the proposal waiting rather than applying it on its own: then it names the loaded packages applying would restart.

`set_date()` returns, invisibly, the applied changes (the same shape as `preview_date()`'s). It is refused (an `ember_refused` condition) unless `preview_date()` for that date is ready and current: the caller must have seen what would change.

`run()` returns, invisibly, the file's exit status.

`clean()` returns, invisibly, a data frame path, kind ("library", "staging", "cache", "index"), bytes, last_used of what was (or with `dry_run`, would be) deleted.

`ember_repos()` returns an `ember_repos` value. `cache_dir()` returns a path: the `ember.cache_dir` option if set, otherwise `tools::R_user_dir("ember", "cache")`.

Examples

```
## Not run:
nb <- open_notebook("growth.R")
edit_notebook(nb, insert_cell(2, "library(dplyr)\n curves |> filter(od > 0.1)"))
package_status(nb)$plan
run_cells(nb, 2, wait = TRUE)
package_status(nb)$library$status

p <- preview_date(nb, Sys.Date())
p$changes
set_date(nb, Sys.Date())
```

```
ember::run("growth.R")
ember::clean(dry_run = TRUE)

## End(Not run)
```

read_cell

Read one cell's code, without a notebook

Description

Statically reads one cell's code: what it defines, what it reads, which packages it attaches or uses, any global settings it changes, and the formulas it builds. Pure: it never runs the code. This is the same reader `notebook_graph()` uses for every cell.

Usage

```
read_cell(code, read_file = NULL)
```

Arguments

code	A single string of R code.
read_file	Function of one path argument, used to follow a literal <code>source()</code> call: it should return the sourced file's text, or NULL if the file can't be read. NULL means <code>source()</code> calls are not followed (their target is recorded as a note instead).

Value

An `ember_cell_analysis`: a list with `code`, `parse_error` (NULL unless the code fails to parse), and data frames definitions, references, packages, settings, sourced, notes, plus a list of `formula_site` in `formulas`. When `parse_error` is set, every other field is empty.

Examples

```
a <- read_cell("fit <- lm(y ~ poly(x, deg), data = df)")
a$definitions$name
a$references$name
a$formulas[[1]]$columns
```

server

*The UI server***Description**

Host Pluto's frontend over HTTP and one websocket per browser tab, backed by any number of ember_notebooks. `start_server()` is for the interactive console: it runs the server in a background process and returns at once. `serve()` is the blocking call behind it, for a host (such as Endeavor) that starts its own R process and wants to open notebooks itself.

Usage

```
serve(paths = character(), port = 0L, secret = random_secret(32),
      on_ready = NULL, launch_browser = FALSE, allowed_hosts = character())

start_server(path = NULL, port = 0L, open = interactive(), timeout = 15,
            allowed_hosts = character())

host_notebook(server, nb, owned = FALSE, remember = FALSE)

stop_server(server)

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

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

Arguments

<code>paths</code>	Notebook files to open in safe preview when <code>serve()</code> starts.
<code>port</code>	Port on 127.0.0.1; 0 tries the stable default, 4321, and the next few ports upward before falling back to a free one at random, so a browser bookmark, an SSH tunnel command (<code>ssh -L 4321:localhost:4321</code>) or a reverse proxy's saved config keeps working across restarts.
<code>secret</code>	The URL secret. Every page, static file request to <code>/edit</code> , and websocket connection needs it (as <code>?secret=</code> or the cookie <code>/edit</code> sets); files served directly from the frontend folder do not.
<code>on_ready</code>	<code>function(server)</code> , called once the port is listening, or <code>NULL</code> . Its <code>server\$port</code> is the port on 127.0.0.1 the server listens on (the one <code>port = 0</code> picked), which a host reads to reach the server. A host opens notebooks with the R API here (or later, from its own callbacks) and hands them to the server with <code>host_notebook()</code> .
<code>launch_browser</code>	Open the first notebook's edit URL (or the server's own URL if none was given) once the server is ready.

<code>allowed_hosts</code>	Host names to accept besides loopback (127.0.0.1, localhost, <code>[::1]</code> , any port), for a reverse proxy that isn't loopback from this R process's point of view – Posit Workbench, JupyterHub's server proxy, a container's exposed port. Each entry can include a port ("ember.example.org:8443", accepted only on that port) or be bare ("ember.example.org", accepted on any port). A request's Origin, when present, must still equal its Host exactly; listing a host here never allows a <i>different</i> origin to connect. Default: none.
<code>path</code>	A notebook file for <code>start_server()</code> to open, or NULL to open the start page instead. New notebooks created from the start page default to the folder this R session started in.
<code>open</code>	Open the notebook (or, with no path, the start page) in the browser once the child is listening.
<code>timeout</code>	Seconds to wait for the child process to report that it is listening.
<code>server</code>	The <code>ember_server</code> value passed to <code>on_ready()</code> .
<code>nb</code>	An <code>ember_notebook</code> , as returned by <code>open_notebook()</code> .
<code>owned</code>	TRUE when the server itself opened nb and should close it when the server stops or the page sends "shut down"; a host's own notebooks are the host's to close, so its calls to <code>host_notebook()</code> leave this FALSE.
<code>remember</code>	Internal: TRUE adds the notebook's path to the start page's recent notebooks. Only Ember's own pages set it (the start page, /open, the paths given to <code>serve()</code> or <code>start_server()</code>); a host leaves it FALSE, so notebooks it drives never show up there as ones the user opened.
<code>x</code>	An <code>ember_server</code> or <code>ember_server_handle</code> , for <code>print()</code> .
<code>...</code>	Unused; present for consistency with <code>print()</code> generics.

Details

`serve()` runs entirely on R's one thread: **httpuv** and the notebook engine both use **later**, so nothing here blocks it. A browser tab's view of a notebook is kept in sync by diffing the engine's current state against what that tab was last sent, after every change and at the end of every request; a page that changes nothing receives nothing.

A server's base URL (/) is its own start page: hosted and recently opened notebooks, "New notebook" and "Open a file". New notebooks default to the folder the server was started in. The recent list is a single file per user under `tools::R_user_dir("ember", "data")`, shared by every Ember server that user runs, whatever the port.

`start_server()` is for interactive use: running **httpuv** in the current R session would freeze it whenever that session is busy running cells, so the server runs in a child process instead, and the secret reaches it through an environment variable rather than a command-line argument other users on the machine could read.

Value

`serve()` returns, invisibly, once `stop_server()` has been called from a callback; it never returns on its own.

`start_server()` prints the link to open (the notebook's, given a path; otherwise the start page's) as a message, and returns, invisibly, an `ember_server_handle`: an environment with `url` (the server's

base URL, including ?secret=), `open(path)` (opens a notebook through /open and returns its edit URL, invisibly), `stop()` (kills the child process; its worker processes die with it), `output` (the child process's recent output lines), and `process` (the **processx** handle). While the child runs, a browser that asks for a page without the current key gets the link printed again in this console.

`host_notebook()` returns the notebook's edit URL (with the secret). If the server already hosts that notebook, nothing changes and the same URL is returned.

`stop_server()` returns NULL, invisibly.

Examples

```
## Not run:
# At the console: the server runs in a child process.
srv <- start_server("growth.R")
srv
srv$open("other.R")
srv$stop()

# Endeavor's host, blocking its own process.
serve(port = 0L, secret = Sys.getenv("ENDEAVOR_EMBER_SECRET"),
      on_ready = function(server) {
        nb <- open_notebook("growth.R")
        url <- host_notebook(server, nb)
        message("edit at: ", url)
      })

## End(Not run)
```

Index

`add_extra_package` (packages), 9
`affected` (graph_queries), 3
`allow_execution` (notebooks), 6

`blocked_cells` (graph_queries), 3

`cache_dir` (packages), 9
`cell_errors` (graph_queries), 3
`cell_summary` (graph_queries), 3
`changed_cells` (graph_queries), 3
`clean` (packages), 9
`close_notebook` (notebooks), 6

`delete_cell` (edit_notebook), 2
`dependency_graph` (notebooks), 6
`disable_cell` (edit_notebook), 2
`downstream` (graph_queries), 3

`edit_notebook`, 2
`ember_repos` (packages), 9

`fold_cell` (edit_notebook), 2
`format.ember_graph_error`
 (graph_queries), 3

`graph_learn` (notebook_graph), 5
`graph_queries`, 3

`host_notebook` (server), 13

`insert_cell` (edit_notebook), 2
`interrupt_notebook` (notebooks), 6

`move_cell` (edit_notebook), 2
`move_notebook` (notebooks), 6

`new_notebook` (notebooks), 6
`notebook_graph`, 3, 5, 8
`notebook_snapshot` (notebooks), 6
`notebook_state` (notebooks), 6
`notebooks`, 6

`on_notebook_event` (notebooks), 6
`open_notebook` (notebooks), 6

`package_status` (packages), 9
`packages`, 7, 8, 9
`preview_date` (packages), 9
`print.ember_graph` (graph_queries), 3
`print.ember_notebook` (notebooks), 6
`print.ember_server` (server), 13
`print.ember_server_handle` (server), 13

`read_cell`, 5, 12
`remove_extra_package` (packages), 9
`render_png` (notebooks), 6
`restart_notebook` (notebooks), 6
`run` (packages), 9
`run_cells` (notebooks), 6
`run_order` (graph_queries), 3

`serve` (server), 13
`server`, 13
`set_cell_change_mode` (notebooks), 6
`set_code` (edit_notebook), 2
`set_date` (packages), 9
`start_server` (server), 13
`stop_server` (server), 13

`upstream` (graph_queries), 3

`wait_for` (notebooks), 6