Package: drogonR (via r-universe)

October 26, 2026

Type Package

Title High-Performance HTTP Server for R via 'Drogon'

Version 0.1.9

Description Provides an 'R' interface to the 'Drogon' high-performance 'C++' 'HTTP' server framework (<https://github.com/drogonframework/drogon>). Offers a 'plumber'-style application programming interface for building 'REST' services from 'R' with substantially higher throughput, including streaming responses and full-duplex 'WebSocket' endpoints.

Depends R (>= 4.1.0)

Imports jsonlite, processx, later (>= 1.4.4)

LinkingTo later

Suggests testthat (>= 3.0.0), httr2, curl, websocket, plumber, knitr, rmarkdown

VignetteBuilder knitr

License MIT + file LICENSE

Encoding UTF-8

SystemRequirements C++17, GNU make, OpenSSL (optional, for HTTPS)

NeedsCompilation yes

RoxygenNote 7.3.3

Config/testthat/edition 3

URL https://github.com/Zabis13/drogonR

BugReports https://github.com/Zabis13/drogonR/issues

Author Yuri Baramykov [aut, cre] (ORCID: <https://orcid.org/0009-0000-7627-4217>), An Tao [ctb, cph] (Author of the bundled Drogon and Trantor C++ libraries), Shuo Chen [ctb, cph] (Author of the Muduo library, on which Trantor is based), Baptiste Lepilleur [ctb, cph] (Original author of the bundled JsonCpp library), Christopher Dunn [ctb]

(Maintainer of JsonCpp), JsonCpp Contributors [ctb, cph] (See src/drogon/third_party/jsoncpp/AUTHORS in the package source), Bert Belder [ctb, cph] (Author of the bundled wepoll library (Windows epoll shim)), mman-win32 contributors [ctb, cph] (Authors of the bundled mman-win32 library; see src/mman-win32/LICENSE)

Maintainer Yuri Baramykov <lbsbmsu@mail.ru>

Config/pak/sysreqs make libssl-dev

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

Date/Publication 2026-08-27 12:00:19 UTC

RemoteUrl https://github.com/cran/drogonR

RemoteRef HEAD

RemoteSha 4ddae76d6439594366e08b1f8701b64883b34ee6

Contents

dr_app . . . 3
dr_body . . . 3
dr_file . . . 4
dr_header . . . 5
dr_html . . . 5
dr_json . . . 6
dr_on_error . . . 7
dr_query . . . 8
dr_rate_limit . . . 8
dr_redirect . . . 9
dr_response . . . 10
dr_routes . . . 11
dr_routes_cpp . . . 12
dr_routes_cpp_stream . . . 13
dr_running . . . 14
dr_serve . . . 15
dr_static . . . 17
dr_status . . . 18
dr_stop . . . 18
dr_stream . . . 19
dr_stream_sse . . . 20
dr_text . . . 21
dr_use . . . 22
dr_ws . . . 23
dr_ws_broadcast . . . 24
dr_ws_client_close . . . 24
dr_ws_client_send . . . 25
dr_ws_close . . . 25
dr_ws_connect . . . 26
dr_ws_cpp . . . 27

dr_ws_join 28
dr_ws_leave 29
dr_ws_send 29
dr_ws_shutdown 30
pr_run 30

Index **32**

`dr_app` *Create a drogonR application*

Description

Creates a fresh, empty ‘drogon_app‘ object that holds the route table and configuration for a server. Routes are added with [dr_get()], [dr_post()], [dr_put()], [dr_delete()], and the server is started with [dr_serve()].

Usage

```
dr_app()
```

Details

The returned object is a mutable [environment] (so route-registration calls modify it in place and return it invisibly for use with ‘|>‘).

Value

An object of class ‘drogon_app‘.

Examples

```
app <- dr_app()
app <- dr_get(app, "/", function(req) "hello")
```

`dr_body` *Read the request body*

Description

Returns the body as raw text, parsed JSON, or a raw byte vector.

Usage

```
dr_body(req, as = c("text", "json", "raw"))
```

Arguments

req	A ‘drogon_request‘.
as	Output form: ‘"text"‘ (default), ‘"json"‘, or ‘"raw"‘. ‘"json"‘ requires the ‘jsonlite‘ package.

Value

A character string, parsed R object, or raw vector.

---

dr_file *Build a file response*

Description

Reads ‘path‘ into memory as raw bytes and returns it as the body. For v0.1 the entire file is held in R memory; sendfile-style zero-copy delivery is planned for v0.2 via ‘dr_static()‘. Files larger than 50 MB emit a warning; files larger than 500 MB raise an error to prevent accidental out-of-memory loads.

Usage

```
dr_file(
  path,
  content_type = NULL,
  status = 200L,
  headers = list(),
  download_as = NULL
)
```

Arguments

path	Path to a regular file readable by the calling process.
content_type	MIME type for the response. ‘NULL‘ (the default) auto-detects from the file extension via a built-in table; unknown extensions become ‘application/octet-stream‘.
status	Integer HTTP status code, default 200.
headers	Named list of additional response headers.
download_as	If a non-empty string, sets ‘Content-Disposition: attachment; filename="..."‘ so browsers prompt to save under that name.

Value

A response list (see [dr_response()]).

Examples

```
## Not run:
dr_file("/tmp/report.pdf")
dr_file("/tmp/report.pdf", download_as = "Q3-report.pdf")

## End(Not run)
```

---

dr_header *Read a request header*

Description

Looks up a header by name, case-insensitively.

Usage

```
dr_header(req, name)
```

Arguments

req	A ‘drogon_request‘ passed to your route handler.
name	Header name (e.g. ‘"Content-Type"‘).

Value

The header value as a single string, or ‘NULL‘ if absent.

---

dr_html *Build an HTML response*

Description

Sets ‘Content-Type: text/html; charset=utf-8‘.

Usage

```
dr_html(body = "", status = 200L, headers = list())
```

Arguments

body	Response body as a character string or raw vector.
status	Integer HTTP status code, default 200.
headers	Named list of additional response headers. An explicit ‘Content-Type‘ here wins over the default.

Value

A response list (see [dr_response()]).

Examples

```
dr_html("<h1>hi</h1>")
```

`dr_json` *Build a JSON response*

Description

Serialises ‘x‘ with [jsonlite::toJSON()] and sets ‘Content-Type: application/json‘ (unless already set in ‘headers‘).

Usage

```
dr_json(x, status = 200L, headers = list(), auto_unbox = TRUE)
```

Arguments

`x`	R object to serialise.
`status`	Integer HTTP status code, default 200.
`headers`	Named list of additional response headers.
`auto_unbox`	Passed to [jsonlite::toJSON()]; default ‘TRUE‘ so length-1 vectors become JSON scalars.

Value

A response list (see [dr_response()]).

Examples

```
dr_json(list(ok = TRUE, n = 1L))
```

`dr_on_error` *Register a custom error handler*

Description

Install a function that builds the response when a route handler or middleware throws an R error. The function receives ‘(req, err)‘ — the request object and the captured ‘condition‘ — and must return a response (string, [dr_response()], [dr_json()], etc.). It is called on the main R thread, after the handler / middleware chain has already failed; returning normally short-circuits the default 500.

Usage

```
dr_on_error(app, fn)
```

Arguments

`app`	A ‘drogon_app‘ created by [dr_app()].
`fn`	A function of two arguments, ‘function(req, err)‘. Pass ‘NULL‘ to clear a previously-registered handler.

Details

If the on-error function itself throws, drogonR logs **both** the original handler error and the on-error error to stderr (via [message()]) and falls back to the default plain-text 500 — the client never sees a hung connection. Only one on-error handler is active per app; calling ‘dr_on_error()‘ again replaces it.

Value

The ‘app‘, invisibly.

Examples

```
app <- dr_app() |>
  dr_on_error(function(req, err) {
    dr_json(list(error = conditionMessage(err),
                 path  = req$path),
            status = 500L)
  }) |>
  dr_get("/boom", function(req) stop("nope"))
```

dr_query *Read query-string parameters*

Description

Returns either the named character vector of all query parameters (when ‘name = NULL‘, the default), or the value of a single parameter. Drogon parses and URL-decodes the query string before delivery.

Usage

```
dr_query(req, name = NULL)
```

Arguments

req	A ‘drogon_request‘.
name	Parameter name, or ‘NULL‘ to get the full named vector.

Value

A named character vector when ‘name‘ is ‘NULL‘, otherwise a single string or ‘NULL‘ if the parameter is absent.

dr_rate_limit *Apply a rate limit to one or more routes*

Description

Adds a rate-limit rule to ‘app‘. On each matching request, the Drogon I/O thread checks the rule’s bucket *before* dispatching to R; if the bucket is empty the request is rejected with HTTP 429 (Too Many Requests) and a ‘Retry-After‘ header. Multiple ‘dr_rate_limit()‘ calls add independent rules — a request must satisfy *all* of them to pass.

Usage

```
dr_rate_limit(
  app,
  capacity,
  window = 60,
  type = c("sliding_window", "fixed_window", "token_bucket"),
  scope = c("per_route", "global"),
  routes = NULL
)
```

Arguments

app	A ‘drogon_app‘ from [dr_app()].
capacity	Maximum number of requests allowed in ‘window‘ seconds (per-bucket; see ‘scope‘). Integer ‘>= 1‘.
window	Time window for the bucket, in seconds. Default ‘60‘.
type	One of ‘"sliding_window"‘ (default — counts requests in the trailing ‘window‘ seconds), ‘"fixed_window"‘ (resets at wall-clock boundaries), or ‘"token_bucket"‘ (constant refill rate with burst capacity).
scope	‘"per_route"‘ (default) gives every matched route its own bucket. ‘"global"‘ makes one bucket shared across all routes matched by this rule.
routes	Either ‘NULL‘ (the default — applies to every registered route) or a character vector of path **prefixes** (e.g. ‘c("/api/", "/stream/")‘). A route matches if its path starts with any of the given prefixes.

Details

Per-IP limiting is intentionally not provided: do that in a reverse proxy (nginx, Caddy, Cloudflare). This API is for shaping load on specific endpoints from the application side.

Call ‘dr_rate_limit()‘ *after* registering routes (so prefix matches resolve correctly) and *before* [dr_serve()].

Value

The ‘app‘, invisibly.

Examples

```
## Not run:
app <- dr_app() |>
  dr_get("/health", function(req) "ok") |>
  dr_get("/api/users", function(req) "users") |>
  # 100 req/min per route under /api/, health excluded
  dr_rate_limit(capacity = 100L, window = 60, routes = "/api/")
dr_serve(app, port = 8080L)

## End(Not run)
```

---

dr_redirect *Build a redirect response*

Description

Sets the ‘Location‘ header and an empty body. Default status is 302 (Found / temporary). Use ‘status = 301L‘ for permanent moves, ‘303L‘ after a POST, or ‘307L‘/‘308L‘ to preserve the request method.

Usage

```
dr_redirect(location, status = 302L, headers = list())
```

Arguments

location	Target URL (absolute or relative).
status	Integer HTTP status code, default 302.
headers	Named list of additional response headers.

Value

A response list (see [dr_response()]).

Examples

```
dr_redirect("/login")
dr_redirect("https://example.com", status = 301L)
```

---

dr_response *Build an HTTP response*

Description

Constructs the list shape that route handlers must return: a ‘status‘, a ‘body‘, and a list of headers. Returning the result of ‘dr_response()‘ is interchangeable with returning a plain list with the same fields.

Usage

```
dr_response(body = "", status = 200L, headers = list())
```

Arguments

body	Response body as a character string or raw vector.
status	Integer HTTP status code, default 200.
headers	Named list of response headers.

Value

A list with elements ‘status‘, ‘body‘, ‘headers‘.

Examples

```
dr_response("ok")
dr_response("not found", status = 404L)
```

`dr_routes` *Register HTTP route handlers*

Description

Register an R function as the handler for a given HTTP method and path. The handler is called for every matching request with a single argument ‘req‘ — a ‘drogon_request‘ object. The handler must return either a single character string (sent as ‘text/plain‘, status 200) or the result of [dr_response()] / [dr_json()].

Usage

```
dr_get(app, path, handler)

dr_post(app, path, handler)

dr_put(app, path, handler)

dr_delete(app, path, handler)
```

Arguments

`app`	A ‘drogon_app‘ created by [dr_app()].
`path`	Request path, e.g. ‘"/users"‘.
`handler`	A function of one argument (the request object).

Details

Routes must be registered *before* calling [dr_serve()]. Each call returns the ‘app‘ invisibly so calls can be chained with ‘|>‘.

Value

The ‘app‘ (modified in place), invisibly.

Examples

```
app <- dr_app()
app <- dr_get(app, "/ping", function(req) "pong")
app <- dr_post(app, "/echo", function(req) req$body)
```

`dr_routes_cpp` *Register a native C / C++ route handler*

Description

Bind a path to a handler implemented in another R package's C / C++ code, looked up via [base::getNativeSymbolInfo()]-style ‘R_RegisterCCallable‘ / ‘R_GetCCallable‘. The handler runs on Drogon's worker thread pool — **never** on the R main thread — so its hot path bypasses the R dispatcher entirely. Use this for inference-bound APIs (embeddings, classifiers, GGML/llama.cpp wrappers) where SEXP allocation and ‘R_tryEval‘ per request would dominate latency.

Usage

```
dr_get_cpp(app, path, package, callable)

dr_post_cpp(app, path, package, callable)

dr_put_cpp(app, path, package, callable)

dr_delete_cpp(app, path, package, callable)
```

Arguments

`app`	A ‘drogon_app‘ created by [dr_app()].
`path`	Request path, with the same ‘:name‘ / ‘<name>‘ / ‘{name}‘ placeholder syntaxes as [dr_get()]. Path parameter values are passed positionally to the handler.
`package`	Name of the backend R package that registered the callable.
`callable`	Name passed to the backend's ‘R_RegisterCCallable("<package>", "<callable>", ...)‘ call.

Details

The handler signature is ‘drogonr_unary_handler_t‘, defined in ‘<drogonR.h>‘ (shipped under ‘inst/include/‘). Backend packages should ‘LinkingTo: drogonR‘ in their DESCRIPTION, ‘#include <drogonR.h>‘ in their C / C++ sources, and call ‘R_RegisterCCallable("<package>", "<callable>", ...)‘ in their ‘R_init_<package>‘ to expose the function.

Lookup is eager: ‘dr_get_cpp()‘ calls ‘requireNamespace(package)‘ immediately and resolves ‘<callable>‘ against the loaded DLL. If the package is not installed or the callable is unregistered, the error fires here (during route registration), not silently at request time.

Value

The ‘app‘, invisibly.

Threading and the R API

Native handlers are invoked on Drogon worker threads. They MUST NOT touch any ‘SEXP‘ or call any function in the R API (‘Rf_*‘, ‘R_*‘, ‘Rprintf‘, etc.) — doing so is undefined behaviour. Load models, allocate caches, and read configuration from R BEFORE [dr_serve()] is called; per-request work runs in pure C / C++.

Middleware and error handler

R-side [dr_use()] middleware and the [dr_on_error()] hook are **not** invoked for native routes — they require the request to enter R, which is exactly what this path avoids. Authentication, logging, header injection, etc. must be done either inside the backend handler itself or in front of drogonR (e.g. in a reverse proxy). Per-route [dr_rate_limit()] rules **are** applied (the check runs on the I/O thread before dispatch).

Examples

```
## Not run:
# In package ggmlR, R_init_ggmlR() does:
#   R_RegisterCCallable("ggmlR", "embed",
#                       (DL_FUNC) ggmlr_embed);
app <- dr_app() |>
  dr_post_cpp("/embed", package = "ggmlR", callable = "embed")
dr_serve(app, port = 8080L)

## End(Not run)
```

---

dr_routes_cpp_stream *Register a streaming native (R-bypass) handler*

Description

Like [dr_get_cpp()] but for streaming responses (HTTP chunked / SSE / LLM token streams). The backend handler runs on a drogonR worker thread and pushes chunks via the C callbacks declared in ‘<drogonR.h>‘ (‘drogonr_stream_handler_t‘).

Usage

```
dr_get_cpp_stream(
  app,
  path,
  package,
  callable,
  content_type = "text/event-stream"
)

dr_post_cpp_stream(
  app,
  path,
```

```
  package,
  callable,
  content_type = "text/event-stream"
)
```

Arguments

app	A ‘drogon_app‘ from [dr_app()].
path	URL path; same syntax as [dr_get()].
package	R package that exposes the handler.
callable	Symbol name registered via ‘R_RegisterCCallable‘.
content_type	Default ‘Content-Type‘ for the response. Defaults to ‘"text/event-stream"‘. The backend may override this per call by writing to ‘*out_content_type‘.

Details

The backend is responsible for ‘R_RegisterCCallable("<package>", "<callable>", ...)‘ with a ‘drogonr_stream_handler_t‘ signature. Mismatched signatures are undefined behavior — there is no runtime type check on the function pointer.

Value

‘app‘, invisibly.

Middleware and error handler

R-side [dr_use()] middleware and the [dr_on_error()] hook are **not** invoked for native streaming routes — the request never enters R. Cross-cutting concerns belong in the backend or in a reverse proxy. Per-route [dr_rate_limit()] rules **are** applied on the I/O thread before the worker is dispatched.

---

dr_running *Is the drogonR server currently running?*

Description

Is the drogonR server currently running?

Usage

```
dr_running()
```

Value

‘TRUE‘ if a server is running in this process, ‘FALSE‘ otherwise.

dr_serve *Start the HTTP server*

Description

Starts the bundled Drogon HTTP server on the given port and number of I/O threads. The Drogon event loop runs in dedicated C++ threads; incoming requests are dispatched to R handlers on the main R thread via [later::later_fd()].

Usage

```
dr_serve(
  app,
  port = 8080L,
  threads = 1L,
  workers = 1L,
  on_worker_start = NULL,
  max_queue = 1024L,
  cpp_workers = 4L,
  upload_path = NULL,
  bandwidth = 0,
  bandwidth_burst = NULL
)
```

Arguments

app	A ‘drogon_app‘ with at least one registered route.
port	TCP port to bind, integer in ‘1..65535‘. Defaults to 8080.
threads	Number of Drogon I/O threads per worker, integer ‘>= 1‘. Defaults to 1.
workers	Number of OS-level worker processes. ‘1L‘ (default) serves in-process. ‘> 1‘ spawns workers as fresh ‘Rscript‘ processes and the calling process becomes a thin supervisor. Not supported on Windows.
on_worker_start	Optional ‘function()‘ run once per worker before its Drogon listener starts. Use it to load models or open per-worker resources. Errors abort that worker (exit status 1).
max_queue	Maximum number of pending requests waiting for an R handler before incoming requests are rejected with HTTP 503 (Service Unavailable). Acts as back-pressure when handlers are slower than the arrival rate, preventing unbounded memory growth. 503 responses are sent directly from a Drogon I/O thread without touching R, so overload has no R-side cost. Default ‘1024L‘.
cpp_workers	Size of the worker thread pool that runs native (R-bypass) handlers registered via ‘dr_*_cpp()‘ and ‘dr_*_cpp_stream()‘. Each in-flight cpp request occupies one thread; streaming handlers hold their thread for the full duration of the response. Default ‘4L‘. Increase if you have many concurrent long-running cpp-stream sessions (e.g. LLM token streams). Has no effect on R-side handlers.

`upload_path` Directory where Drogon stores uploaded files. Defaults to 'NULL', in which case a fresh subdirectory inside [tempdir()] is created so the package never writes to the user's home filespace or the installation directory. Pass an explicit path to override.

`bandwidth` Egress limit in bytes per second applied to **each** connection independently (not a server-wide total). '0', the default, disables shaping. Shaping covers every response body — including static files, chunked streams and WebSocket frames — and is enforced on Drogon's I/O thread with no R involvement. Under TLS the limit applies to the plaintext, so bytes on the wire exceed it slightly by the size of the TLS record framing.

`bandwidth_burst`

Token-bucket capacity in bytes, i.e. how much an idle connection may send at full speed before being held to 'bandwidth'. Defaults to '2 * bandwidth'. Values below one MTU are raised to 1500 bytes internally.

Details

When 'workers > 1', drogonR spawns 'workers' fresh R processes via 'Rscript' (not 'fork()'); each worker runs its own Drogon listener on the same port (Linux/macOS use 'SO_REUSEPORT' for kernel-side load balancing). The calling process is a thin **supervisor** — it does not serve requests itself, only tracks worker pids and reaps them at [dr_stop()]. 'on_worker_start' runs in each worker immediately before its Drogon listener starts, so per-worker state (models, caches) is loaded before the first request lands. Going through 'Rscript'+'exec' (rather than 'parallel::mcparallel()') costs ~200ms of startup per worker but gives each worker a clean R: no inherited sink stack, no inherited 'later' event-loop fds, no half-initialised C++ globals from the supervisor.

If 'on_worker_start' throws in a child, that child exits with status 1 after writing the error to stderr; the supervisor notices it on the next [dr_status()] call and continues with the surviving workers. There is no auto-restart in v0.1.

Value

'NULL', invisibly. Prints a one-line listening message.

Lifetime

Drogon's event loop cannot be restarted in the same R session. After calling [dr_stop()], a new [dr_serve()] in the same process will raise an error — start a fresh R session instead.

Examples

```
## Not run:
app <- dr_app() |>
  dr_get("/hello", function(req) "hi")
dr_serve(app, port = 8080L)

# Multi-process, each worker loads its own model copy
dr_serve(app, port = 8080L, workers = 4L,
         on_worker_start = function() {
           model <<- readRDS("model.rds")
```

```
          })

## End(Not run)
```

`dr_static` *Mount a directory as static files*

Description

Serve every file under ‘dir‘ at URLs starting with ‘mount‘. The files are streamed by Drogon directly from a C++ I/O thread (R is never invoked), so this path supports ‘Range‘ requests and auto-detects ‘Content-Type‘. Both ‘GET‘ and ‘HEAD‘ are accepted; missing files return 404, attempted path traversal returns 403.

Usage

```
dr_static(app, mount, dir)
```

Arguments

`app`	A ‘drogon_app‘ created by [dr_app()].
`mount`	URL prefix to mount under, e.g. ‘"/assets"‘. Must start with ‘/‘. A trailing ‘/‘ is stripped.
`dir`	Local directory to serve from. Must exist at [dr_serve()] time.

Value

The ‘app‘, invisibly.

Path traversal

The handler resolves the requested path against ‘dir‘ and rejects (HTTP 403) any request whose normalised target escapes ‘dir‘ — a ‘..‘ segment, an absolute path, or anything else that would otherwise let a remote caller read files outside the mount.

Middleware and error handler

Static files are served entirely from a C++ I/O thread, so R-side [dr_use()] middleware and the [dr_on_error()] hook do **not** apply — they only run for requests that enter R. If you need authentication, custom headers, or per-file logging on assets, put a reverse proxy in front of drogonR or expose the files through a regular [dr_get()] handler instead.

Examples

```
## Not run:
app <- dr_app() |>
  dr_static("/assets", "./public") |>
  dr_get("/api/ping", function(req) "pong")
dr_serve(app, port = 8080L)
# GET /assets/logo.png streams ./public/logo.png from C++.

## End(Not run)
```

`dr_status` *Status of forked worker processes*

Description

Reports which workers forked by [dr_serve()] are still alive. Polls only when called — there is no background supervisor in v0.1, so dead workers are noticed only here or at [dr_stop()] time. Returns an empty data frame in single-process mode.

Usage

```
dr_status()
```

Value

A data frame with columns ‘pid‘ (integer) and ‘alive‘ (logical), one row per tracked worker child.

`dr_stop` *Stop the HTTP server*

Description

Stops the in-process Drogon event loop (when ‘workers == 1L‘) and joins the I/O threads. In supervisor mode (‘workers > 1L‘), sends ‘SIGTERM‘ to every tracked worker, waits up to ~2s for them to exit, then ‘SIGKILL‘s any survivor. No-op if no server is running and no workers are tracked.

Usage

```
dr_stop()
```

Details

Drogon cannot be restarted in the same R session — see [dr_serve()].

Value

‘NULL‘, invisibly.

`dr_stream` *Open a streaming HTTP response*

Description

Return value for a route handler when the response should be streamed (HTTP chunked transfer). Instead of producing one body string, the handler returns a ‘drogon_stream‘ describing how to generate chunks on demand. The dispatcher pumps ‘next_chunk()‘ on the main R thread, one chunk per pump, until it signals ‘done‘.

Usage

```
dr_stream(
  next_chunk,
  state = NULL,
  content_type = "text/event-stream",
  headers = list(),
  min_interval = 0
)
```

Arguments

`next_chunk`	Function ‘function(state, cancelled)‘ returning ‘list(chunk = , state = , done =)‘. See above.
`state`	Initial state passed to the first pump. Anything an R value can hold; opaque to the dispatcher.
`content_type`	MIME type for the response. Defaults to ‘"text/event-stream"‘ since SSE is the most common use case.
`headers`	Named list of additional response headers to send in the initial chunked-transfer response (status is always 200). ‘Content-Type‘ here overrides the ‘content_type‘ argument.
`min_interval`	Minimum delay in seconds between consecutive ‘next_chunk()‘ calls. ‘0‘ (default) pumps as fast as the event loop allows. Set to ‘0.1‘ to throttle to ~10 chunks/sec, etc. Useful for SSE feeds that should be paced rather than bursted. The delay is a floor, not a guarantee — heavy R-side work or other queued callbacks may push the next pump out further.

Details

Each pump receives the current ‘state‘ and a ‘cancelled‘ flag. ‘cancelled‘ is ‘TRUE‘ when the dispatcher has detected that the client connection is gone; the generator will be invoked exactly once with ‘cancelled = TRUE‘ so it can free state, and the stream is then closed regardless of what the call returns. It returns a list with three slots:

* ‘chunk‘ — character(1) bytes to send right now (sent verbatim; format SSE / NDJSON / etc. yourself, or use one of the helpers built on top of ‘dr_stream()‘). * ‘state‘ — the value passed to the next pump. Pass back the incoming ‘state‘ unchanged if you don’t need to mutate it. * ‘done‘ — ‘TRUE‘ to close the response after this chunk; ‘FALSE‘ to schedule another pump.

Value

A list of class ‘drogon_stream‘ carrying ‘next_chunk‘, ‘state‘, ‘content_type‘, ‘headers‘, and ‘min_interval‘. Return it from a route handler; the dispatcher recognises the class and opens an HTTP chunked-transfer response, then pumps ‘next_chunk()‘ on the main R thread until it signals ‘done = TRUE‘.

Threading

‘next_chunk()‘ always runs on the main R thread. R is single-threaded, so this is the only place it could safely run. Heavy work inside one pump blocks every other request and every other stream until it returns — keep each step short, and split long generation across many pumps.

Examples

```
## Not run:
app <- dr_app() |>
  dr_get("/sse", function(req) {
    dr_stream(
      state      = list(i = 0L, n = 5L),
      next_chunk = function(state, cancelled) {
        if (cancelled || state$i >= state$n) {
          return(list(chunk = "", state = state, done = TRUE))
        }
        state$i <- state$i + 1L
        list(chunk = sprintf("data: %d\n\n", state$i),
             state = state, done = FALSE)
      })
  })
dr_serve(app, port = 8080L)
# curl -N http://127.0.0.1:8080/sse

## End(Not run)
```

`dr_stream_sse` *Open a Server-Sent-Events streaming response*

Description

Convenience wrapper around [dr_stream()] for the common case of an SSE feed where each tick emits one ‘data:‘ field. The generator returns ‘data‘ (a string), ‘state‘, and ‘done‘; the helper formats the SSE frame, splitting embedded newlines into multiple ‘data:‘ lines per the SSE spec, and adds the headers a typical SSE client expects (no caching, no proxy buffering).

Usage

```
dr_stream_sse(generator, state = NULL, headers = list(), min_interval = 0)
```

Arguments

generator	Function ‘function(state, cancelled)‘ returning ‘list(data = , state = , done =)‘. ‘data‘ may contain newlines; they are split into multiple ‘data:‘ lines automatically. An empty ‘data‘ is allowed (sends a keep-alive frame).
state	Initial state, as in [dr_stream()].
headers	Extra response headers to merge with the SSE defaults (‘Content-Type: text/event-stream‘, ‘Cache-Control: no-cache‘, ‘X-Accel-Buffering: no‘). User-supplied entries with the same name win.
min_interval	Floor on the delay between consecutive ‘generator()‘ calls, in seconds. See [dr_stream()] for details. Default ‘0‘ (no throttling).

Details

For SSE features beyond plain ‘data:‘ (‘event:‘, ‘id:‘, ‘retry:‘), use [dr_stream()] directly and format the frame yourself.

Value

A ‘drogon_stream‘ value to return from a route handler.

Examples

```
## Not run:
app <- dr_app() |>
  dr_get("/sse", function(req) {
    dr_stream_sse(
      state = list(i = 0L, n = 5L),
      generator = function(state, cancelled) {
        if (cancelled || state$i >= state$n) {
          return(list(data = "", state = state, done = TRUE))
        }
        state$i <- state$i + 1L
        list(data = sprintf("tick %d", state$i),
             state = state, done = FALSE)
      })
  })
dr_serve(app, port = 8080L)
# curl -N http://127.0.0.1:8080/sse

## End(Not run)
```

---

dr_text *Build a plain-text response*

Description

Sets ‘Content-Type: text/plain; charset=utf-8‘. The charset is explicit because some intermediaries / older clients otherwise fall back to a non-UTF-8 default and mangle non-ASCII bodies.

Usage

```
dr_text(body = "", status = 200L, headers = list())
```

Arguments

body	Response body as a character string or raw vector.
status	Integer HTTP status code, default 200.
headers	Named list of additional response headers. An explicit ‘Content-Type‘ here wins over the default.

Value

A response list (see [dr_response()]).

Examples

```
dr_text("hello")
dr_text("not found", status = 404L)
```

---

dr_use *Register middleware*

Description

Append a middleware function to the app's middleware chain. Middleware runs in registration order before the matched route handler. Each middleware receives ‘(req, nxt)‘: call ‘nxt()‘ to pass control to the next link (its return value is the downstream response, which you may return as-is or modify), or return a response of your own to short- circuit the chain. Throwing an error has the same effect as the route handler throwing — the chain stops and a 500 is returned.

Usage

```
dr_use(app, middleware)
```

Arguments

app	A ‘drogon_app‘ created by [dr_app()].
middleware	A function of two arguments, ‘function(req, nxt)‘.

Value

The ‘app‘, invisibly.

Examples

```
app <- dr_app() |>
  dr_use(function(req, nxt) {
    t0 <- Sys.time()
    res <- nxt()
    message("served ", req$path, " in ",
            format(Sys.time() - t0))
    res
  }) |>
  dr_get("/ping", function(req) "pong")
```

`dr_ws` *Register a WebSocket endpoint*

Description

Adds a full-duplex WebSocket route to a ‘drogon_app‘. Unlike the request/response routes (‘dr_get()‘ etc.), a WebSocket connection is long-lived: the server can push messages to the client at any time via [dr_ws_send()], and the client can send messages back, each delivered to ‘on_message‘.

Usage

```
dr_ws(app, path, on_message, on_connect = NULL, on_close = NULL)
```

Arguments

`app`	A ‘drogon_app‘ (see [dr_app()]).
`path`	Exact request path to serve, e.g. ‘"/ws/echo"‘.
`on_message`	‘function(conn, msg, binary)‘ called for every frame received from the client. ‘binary‘ is ‘TRUE‘ for binary frames. Required.
`on_connect`	‘function(conn)‘ called once when a client connects. Optional.
`on_close`	‘function(conn)‘ called once when the connection closes (client- or server-initiated). Optional.

Details

The three hooks all run on the main R thread (never on Drogon’s I/O threads), so they may touch R state freely. Each receives a ‘drogon_ws_conn‘ handle identifying the connection; pass it to [dr_ws_send()] / [dr_ws_close()].

Value

The ‘app‘, invisibly modified, for piping.

See Also

[dr_ws_send()], [dr_ws_close()]

dr_ws_broadcast *Broadcast a message to every connection in a room*

Description

Sends ‘msg‘ to all connections currently in ‘room‘. Connections that have closed are skipped silently. Safe to call from any ‘dr_ws()‘ hook.

Usage

```
dr_ws_broadcast(room, msg, binary = FALSE)
```

Arguments

room	A single string naming the room.
msg	A single string to send.
binary	Send as a binary frame instead of text. Default ‘FALSE‘.

Value

The number of connections the message was queued to, invisibly.

See Also

[dr_ws_join()], [dr_ws()]

dr_ws_client_close *Close a client WebSocket connection*

Description

Closes a connection opened with [dr_ws_connect()]. The ‘on_close‘ hook (if any) fires once the close completes. A no-op if already closed.

Usage

```
dr_ws_client_close(client)
```

Arguments

client	A ‘drogon_ws_client‘ from [dr_ws_connect()].

Value

‘NULL‘, invisibly.

See Also

[dr_ws_connect()], [dr_ws_shutdown()]

`dr_ws_client_send` *Send a message on a client WebSocket connection*

Description

Pushes a message to the server over a connection opened with [dr_ws_connect()]. If the connection is not (yet) established or has closed, the call is a no-op and returns ‘FALSE‘.

Usage

```
dr_ws_client_send(client, msg, binary = FALSE)
```

Arguments

`client`	A ‘drogon_ws_client‘ from [dr_ws_connect()].
`msg`	A single string to send.
`binary`	Send as a binary frame instead of text. Default ‘FALSE‘.

Value

‘TRUE‘ if the message was queued, ‘FALSE‘ otherwise, invisibly.

See Also

[dr_ws_connect()]

`dr_ws_close` *Close a WebSocket connection*

Description

Initiates a server-side close of a live WebSocket connection. The ‘on_close‘ hook (if any) fires once the close completes. A no-op if the connection is already gone.

Usage

```
dr_ws_close(conn)
```

Arguments

`conn`	A ‘drogon_ws_conn‘ handle, as passed to a ‘dr_ws()‘ hook.

Value

'NULL', invisibly.

See Also

[dr_ws()]

`dr_ws_connect` *Connect to an external WebSocket server*

Description

Opens an outbound WebSocket connection from R to a remote server (for example, a streaming LLM API or a chat backend). Unlike [dr_ws()], which serves inbound connections on a running drogonR server, this is drogonR acting as a *client*, and it works independently of [dr_serve()] — you do not need a running server to use it.

Usage

```
dr_ws_connect(url, on_message, on_open = NULL, on_close = NULL)
```

Arguments

`url`	WebSocket URL, 'ws://host[:port][/path]' or 'wss://host[:port][/path]'. 'wss://' requires a drogonR built with OpenSSL.
`on_message`	'function(msg, binary)' called for every frame received from the server. 'binary' is 'TRUE' for binary frames. Required.
`on_open`	'function()' called once when the connection is established. Optional.
`on_close`	'function(reason, code)' called once when the connection ends. 'reason' is '"connect_failed"' if the initial connect never succeeded, or '"closed"' if an established connection later dropped. Optional.

Details

The connection runs on a dedicated background event loop. All three callbacks fire on the main R thread (never on that loop), so they may touch R state freely. The first connection lazily starts the client subsystem; it stays alive until [dr_ws_shutdown()] or package unload, so an open/close cycle does not thrash the background thread.

Value

A 'drogon_ws_client' handle; pass it to [dr_ws_client_send()] and [dr_ws_client_close()].

See Also

[dr_ws_client_send()], [dr_ws_client_close()], [dr_ws_shutdown()]

dr_ws_cpp *Register a C++ (R-bypass) WebSocket endpoint*

Description

Like [dr_ws()], but every frame is handled by a compiled C function in another R package rather than by R closures. The handler runs on Drogon's I/O thread with no round-trip through R, which is what you want when a backend owns the socket and streams data from C/C++ (e.g. an LLM emitting tokens).

Usage

```
dr_ws_cpp(
  app,
  path,
  package,
  callable,
  max_conns = 0,
  idle_timeout = 0,
  max_lifetime = 0
)
```

Arguments

app	A ‘drogon_app‘ (see [dr_app()]).
path	Exact request path to serve.
package	Name of the package exporting the callable.
callable	Name of the ‘R_RegisterCCallable()‘ symbol.
max_conns	Maximum concurrent connections on this route; ‘0‘ (default) for unlimited.
idle_timeout	Seconds without an outgoing frame before the connection is closed; ‘0‘ (default) disables idle reaping.
max_lifetime	Absolute seconds a connection may live; ‘0‘ (default) disables the lifetime ceiling.

Details

The ‘callable‘ must be a function pointer of type ‘drogonr_ws_handler_t‘ (see ‘inst/include/drogonR.h‘), registered by the backend package via ‘R_RegisterCCallable()‘. It is resolved eagerly here, so a missing symbol fails at registration, not on the first connection.

A C++ route ignores the R hooks of [dr_ws()]; the two cannot be mixed on the same path.

Continuous-batching guards

The optional ‘max_conns‘, ‘idle_timeout‘, and ‘max_lifetime‘ bound a native route the way ‘-np N‘ and the slot timeouts bound ‘llama-server‘. They exist so an LLM scheduler layered on this ABI doesn't have to re-implement them, less reliably, itself:

* ‘max_conns‘ — refuse a new connection once this many are already live on the route. The rejected client gets a WebSocket close with code 1013 ("Try Again Later"), not a bare TCP reset, so it can back off and retry. ‘0‘ (default) means unlimited. * ‘idle_timeout‘ — close a connection after this many seconds with no outgoing frame (i.e. the backend has stopped streaming tokens to it — typically a stalled or crashed decode thread). ‘0‘ disables. * ‘max_lifetime‘ — hard ceiling, in seconds, on a connection’s total duration regardless of activity. ‘0‘ disables.

All three apply only to C++ routes; they are the transport’s safety net, not a substitute for the backend’s own queue bound.

Value

The ‘app‘, invisibly modified, for piping.

See Also

[dr_ws()]

`dr_ws_join` *Add a connection to a broadcast room*

Description

Rooms are named groups of WebSocket connections. [dr_ws_broadcast()] sends a message to every member of a room at once, so you don’t have to track connection handles yourself. Joining a room a connection is already in is a no-op.

Usage

```
dr_ws_join(conn, room)
```

Arguments

`conn`	A ‘drogon_ws_conn‘ handle, as passed to a ‘dr_ws()‘ hook.
`room`	A single string naming the room.

Details

A connection is removed from all its rooms automatically when it closes, so you only need [dr_ws_leave()] to remove it early.

Value

‘NULL‘, invisibly.

See Also

[dr_ws_broadcast()], [dr_ws_leave()], [dr_ws()]

dr_ws_leave *Remove a connection from a broadcast room*

Description

Remove a connection from a broadcast room

Usage

```
dr_ws_leave(conn, room)
```

Arguments

conn	A ‘drogon_ws_conn‘ handle, as passed to a ‘dr_ws()‘ hook.
room	A single string naming the room.

Value

‘NULL‘, invisibly.

See Also

[dr_ws_join()], [dr_ws_broadcast()]

dr_ws_send *Send a message on a WebSocket connection*

Description

Pushes a message to the client of a live WebSocket connection. Safe to call from any of the ‘dr_ws()‘ hooks. If the connection has already closed, the call is a silent no-op.

Usage

```
dr_ws_send(conn, msg, binary = FALSE)
```

Arguments

conn	A ‘drogon_ws_conn‘ handle, as passed to a ‘dr_ws()‘ hook.
msg	A single string to send.
binary	Send as a binary frame instead of text. Default ‘FALSE‘.

Value

‘NULL‘, invisibly.

See Also

[dr_ws()]

`dr_ws_shutdown` *Shut down the WebSocket client subsystem*

Description

Closes every open client connection and stops the background event loop that services them. Called automatically when the package is unloaded; call it explicitly to release the background thread early. After shutdown, a fresh [dr_ws_connect()] transparently restarts the subsystem.

Usage

```
dr_ws_shutdown()
```

Value

'NULL', invisibly.

See Also

[dr_ws_connect()]

`pr_run` *Run a plumber router under drogonR (drop-in shim)*

Description

Translate a [plumber::pr()] router into [dr_app()] routes and start the drogonR server. The intent is a one-line replacement: existing 'plumber::pr_run(pr)' becomes 'drogonR::pr_run(pr)' without further changes, for the subset of plumber that drogonR can faithfully reproduce.

Usage

```
pr_run(pr, host = "0.0.0.0", port = 8080L, ...)
```

Arguments

pr	A ‘Plumber‘ router created by [plumber::pr()].
host	Host to bind. Only ‘"0.0.0.0"‘, ‘"127.0.0.1"‘, ‘"localhost"‘, and ‘"::"‘ are accepted; anything else triggers a warning and binds to ‘0.0.0.0‘ (drogonR always binds to the wildcard).
port	TCP port, integer in ‘1..65535‘.
...	Additional arguments forwarded to [dr_serve()] (e.g. ‘threads‘, ‘workers‘, ‘max_queue‘). Plumber-specific arguments that have no analogue in drogonR (‘docs‘, ‘swagger‘, ‘swaggerCallback‘, ‘quiet‘) are silently accepted and ignored, so that an existing ‘plumber::pr_run(pr, docs = FALSE)‘ call site keeps working after swapping in the shim.

Value

‘NULL‘, invisibly. Blocks the calling thread until the drogonR server is stopped from another R session via [dr_stop()] (this matches ‘plumber::pr_run()‘ semantics).

Supported

* ‘@get‘, ‘@post‘, ‘@put‘, ‘@delete‘ annotations. * Path placeholders ‘<name>‘ and ‘<name:type>‘. Recognised types are ‘int‘/‘integer‘, ‘dbl‘/‘double‘/‘numeric‘, ‘bool‘/‘logical‘; anything else is left as a character. Coercion runs only on path parameters (plumber does not coerce query / body args at the serializer layer either). * Handler arguments resolved by name from path > query > JSON body, with ‘req‘ injected if the handler takes a ‘req‘ parameter. * Default plumber serialisation: every return value goes through ‘jsonlite::toJSON(auto_unbox = FALSE)‘ – including bare strings, which become JSON arrays – for byte-level parity with ‘plumber::pr_run()‘. Returning a [dr_response()] / [dr_json()] / etc. opts out and is forwarded as-is.

Not supported

Filters (‘@filter‘), hooks (‘pr_hook()‘), mounts (‘pr_mount()‘), custom parsers/serialisers, OpenAPI assets, websockets, async handlers, and the ‘res‘ (response) parameter of plumber handlers. Each of these triggers an explicit error or per-route warning at ‘pr_run()‘ time so failure is loud, not silent. If you need any of these, use the native [dr_app()] API.

Index

dr_app, 3
dr_body, 3
dr_delete *(dr_routes)*, 11
dr_delete_cpp *(dr_routes_cpp)*, 12
dr_file, 4
dr_get *(dr_routes)*, 11
dr_get_cpp *(dr_routes_cpp)*, 12
dr_get_cpp_stream
(dr_routes_cpp_stream), 13
dr_header, 5
dr_html, 5
dr_json, 6
dr_on_error, 7
dr_post *(dr_routes)*, 11
dr_post_cpp *(dr_routes_cpp)*, 12
dr_post_cpp_stream
(dr_routes_cpp_stream), 13
dr_put *(dr_routes)*, 11
dr_put_cpp *(dr_routes_cpp)*, 12
dr_query, 8
dr_rate_limit, 8
dr_redirect, 9
dr_response, 10
dr_routes, 11
dr_routes_cpp, 12
dr_routes_cpp_stream, 13
dr_running, 14
dr_serve, 15
dr_static, 17
dr_status, 18
dr_stop, 18
dr_stream, 19
dr_stream_sse, 20
dr_text, 21
dr_use, 22
dr_ws, 23
dr_ws_broadcast, 24
dr_ws_client_close, 24
dr_ws_client_send, 25
dr_ws_close, 25
dr_ws_connect, 26
dr_ws_cpp, 27
dr_ws_join, 28
dr_ws_leave, 29
dr_ws_send, 29
dr_ws_shutdown, 30

pr_run, 30