Package: freshwater (via r-universe)

May 15, 2026

Title Server-side rendering utilities for Plumber2 APIs

Version 0.0.0.9000

Description Provides various utility functions for improving server-side rendered HTML applications, such as enhanced HTML templating, response caching, and HTML tag serialisation.

License GPL (>= 3)

Encoding UTF-8

Roxygen list(markdown = TRUE)

RoxygenNote 7.3.3.9000

Imports htmltools, memoise, plumber2, rlang, cachem, glue, waysign, stats, utils, stringi, cli, mirai, mori, promises

Suggests fiery, openssl, testthat (>= 3.0.0), later, otel

Config/testthat/edition 3

URL https://elianhugh.github.io/freshwater/

Collate 'cget.R' 'handler_hooks.R' 'context.R' 'security.R' 'template.R' 'errors.R' 'serialisers.R' 'template_cache.R' 'url.R' 'zzz.R'

Config/pak/sysreqs cmake libfontconfig1-dev libfreetype6-dev libfribidi-dev make libharfbuzz-dev libicu-dev libjpeg-dev libpng-dev libsodium-dev libtiff-dev libuv1-dev libwebp-dev libxml2-dev libssl-dev libx11-dev xz-utils zlib1g-dev libclang-dev

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

Date/Publication 2026-05-15 01:06:24 UTC

RemoteUrl https://github.com/ElianHugh/freshwater

RemoteRef HEAD

RemoteSha eb4f187f65c99f1845c12901f4a89093382cfa91

Contents

`api_cget` *Conditional GET*

Description

Creates a conditional GET handler for a specific HTTP path, using a supplied etag function for the current representation. If the server's ETag and client's 'If-None-Match' headers match, a `304 Not Modified` response is sent, short-circuiting downstream handlers. This reduces the need to recompute responses for paths where the data has not changed since the last client request.

Usage

```
api_cget(api, path, etag_fn)
```

Arguments

`api`	a plumber2::plumber2 api object.
`path`	the path to short circuit.
`etag_fn`	a function that takes either zero or one argument, and returns a single value used to derive the ETag.

Details

See https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Conditional_requests for more information.

Annotation Reference

The etag function is specified by `@etag fn` where `fn` is the function name. Functions can also be defined in-line like `@etag \() x + 1`.

```
increment_x <- \() {
    x <<- x + 1L
    later::later(increment_x, delay = 10L)
}
x <- 1L
increment_x()
#* @get /
#* @etag \() x
function() {
 x
}
```

`api_csrf` *Apply CSRF Protection to a plumber2 API*

Description

`api_csrf()` installs CSRF middleware on a plumber2 API using the double-submit cookie pattern.

When installed:

- Any `form` element inside template automatically includes a CSRF token.
- If working in JavaScript contexts, the `csrf_token()` helper is also accessible inside templates.

Middleware behaviour:

- On **safe** methods (`GET`, `HEAD`, `OPTIONS`), if the CSRF cookie is missing, a new token is generated and set as a cookie.
- On **unsafe** methods (`POST`, `PUT`, `DELETE`, `PATCH`), the request is rejected with **403 Forbidden** unless a token provided via the `X-CSRF-Token` header or a `csrf_token` field in the parsed request body matches the CSRF cookie.

This middleware installs freshwater request context.

Usage

```
api_csrf(api, secure = TRUE, exemptions = character())
```

Arguments

`api`	a plumber2 API object
`secure`	if `TRUE`, sets the CSRF cookie to "__Host-csrf" and marks the cookie as secure. If false, uses "csrf".
`exemptions`	character vector of route patterns to exempt from CSRF checks

Annotation Reference

Method-scoped CSRF exemptions can be specified by `@csrf`:

- `"on"`: (default) CSRF checks are enforced
- `"off"` or `"exempt"`: CSRF checks are skipped for the route

```
#* @post /foo/*/bar
#* @csrf exempt
function() {
 print("No checking!")
}
```

Background

Cross-site request forgery (CSRF) refers to attacks that trick user browsers into making unintended unsafe HTTP requests to trusted sites – often through piggybacking on existing authenticated user sessions.

In general, clients are most vulnerable when only cookies are used to validate requests from authenticated users. Read more: https://developer.mozilla.org/en-US/docs/Web/Security/Attacks/CSRF

See Also

form, api_freshwater, api_hooks

Examples

```
#* @plumber
function(api) {
  api |>
       api_csrf(secure = FALSE, exemptions = c("/foo/*", "/bar"))
}
```

---

api_error_pages *Freshwater Error Pages*

Description

Adds request/error hooks to a plumber2::plumber2 API so that freshwater can render friendly **HTML error pages** for:

- **403 Forbidden** responses
- **404 Not Found** responses
- **500 Internal Server Error** conditions

Usage

```
api_error_pages(
  api,
  handlers = NULL,
  debug = plumber2::get_opts("fw_debug", default = interactive())
)
```

Arguments

api	a plumber2::plumber2 api object.
handlers	optional list of named error templates. Supported keys are: "403", "404", "500". If omitted, freshwater installs default templates.
debug	whether the **500** error template should render error messages and stack traces. Defaults to the fw_debug plumber2 option, and falls back to interactive().

Details

Custom error page templates can be supplied via the handlers parameter. These should be freshwater templates created via template(), and should match the call signatures of the default error templates. See freshwater_error_templates for the relevant template signatures required.

This middleware installs freshwater request context.

See Also

freshwater_error_templates, api_hooks

Examples

```
#* @plumber
function(api) {
 api |>
     api_error_pages(debug = TRUE)
}
```

---

api_freshwater *Freshwater defaults for plumber2 APIs*

Description

Installs freshwater defaults onto a plumber2 API.

This is a convenience wrapper:

- Registers freshwater's HTML serialiser
- Registers freshwater's async evaluator
- Installs freshwater request context

- Optionally enables CSRF protection
- Optionally installs HTML error page handlers

Arguments in ... are selectively forwarded to `api_csrf()`, `api_error_pages()`, `register_async_evaluator()`, and `register_html_serialiser()` based on matching formal parameters.

Usage

```
api_freshwater(api, csrf = TRUE, error_pages = TRUE, ...)
```

Arguments

`api`	a plumber2::plumber2 api object.
`csrf`	whether to enable CSRF protection
`error_pages`	whether to enable error pages
`...`	args passed to `api_csrf()`, `api_error_pages()`, `register_html_serialiser()`, or `register_async_evaluator()`

See Also

api_csrf, api_error_pages, register_html_serialiser, register_async_evaluator

---

`api_hooks` *Route handler hooks*

Description

Add middleware-style hooks to *all existing user handlers* in a plumber2::plumber2 API. Hooks execute in a deterministic order prior to the user handler, and can mutate/intercept requests and responses, as well as short-circuit handlers. This allows for middleware-type behaviour without registering additional routes.

Usage

```
api_hooks(api, hooks, .where = c("append", "prepend"))

hook(id = NULL, fn)
```

Arguments

`api`	a plumber2::plumber2 api object.
`hooks`	a single hook or list of hooks that take the signature `fn(api, args, next_call)`, where `args` is the list of handler arguments. If a hook returns without calling `next_call()`, the remaining hooks and user handler are skipped, and the return value becomes the handler result.
`.where`	whether the hooks should be appended or prepended to the list of installed hooks
`id`	id of the hook
`fn`	function with signature `fn(api, args, next_call)`

Details

System hooks are installed in the following order:

- `freshwater::context`
- `freshwater::error_pages`
- `freshwater::csrf`
- `freshwater::csrf_context`

Control Flow:

Hooks control whether subsequent hooks should execute. Concretely:

- To continue to the next hook (and eventual user handler), call `next_call()`.
- To short-circuit the chain, return a value without calling `next_call()`.
- To bubble up to plumber2 routing control flow, return either `plumber2::Next` or `plumber2::Break` (and don't call `next_call()`).

Hooks can also wrap later hooks and the user handler by calling `next_call()`, and then doing work after.

Hook Installation:

- Routing is managed by plumber2::plumber2 – this function does not change routing precedence. Within the handler itself, however, hooks run in the order they are installed.
- The function is idempotent (with respect to either a computed hash of the hook or a provided id), and only new hooks will be installed.

Asynchronous Routes:

When using asynchronous routes via `async=TRUE` programmatically, or via `@async`, hooks are attached to the `then` handlers, rather than main handler itself. This is because `request`, `response`, and `server` arguments are not available to the main async handler, and hooks depend on the full handler signature being available.

Examples

```
api <- plumber2::api() |>
  plumber2::api_get(path = "/", function() {
    "Foo"
  })

log_hook <- hook(
  "logger",
  function(api, args, next_call) {
    msg <- sprintf(
      "[%s] %s %s",
      format(Sys.time(), "%H:%M:%S"),
      args$request$method,
      args$request$path
    )
    print(msg)
    next_call()
  }
```

```
)

timer_hook <- hook("timer", function(api, args, next_call) {
  t0 <- Sys.time()
  out <- next_call()
  print(sprintf("time: %s", Sys.time() - t0))
  out
})
api <- api_hooks(api, hooks = list(log_hook, timer_hook))
```

---

cache *Cache a partial within a template*

Description

`cache()` memoises a portion of a template as an HTML tag subtree. The contents are computed once per unique cache key, and reused in subsequent calls. This avoids repeat evaluation of expensive or stable HTML trees.

`clear_cache()` removes all memoised templates from freshwater's cache store.

`invalidate_cache()` removes a single cached entry identified by `name`, and optionally via `vary` and fragment values. Note that the `invalidate_cache` arguments must match those in the original `cache` call, as they are used to construct the cache key.

`invalidate_cache_here()` is the in-template version of `invalidate_cache`. It uses the current template execution context to allow users to forcibly regenerate caches inside the template function.

Usage

```
cache(name, vary = NULL, ttl = NULL, ...)

clear_cache()

invalidate_cache(tpl, name, vary = NULL, fragment = NULL)

invalidate_cache_here(name, vary = NULL, fragment = NULL)
```

Arguments

`name`	unique name for the cached partial template
`vary`	values that should change when the cached output should change. This is used to construct the cache key.
`ttl`	when the cache should expire. When NULL, will only expire when the cache is invalidated.
`...`	tag content to render and cache
`tpl`	a template function created by `template()`.
`fragment`	optional fragment name for targetting cached fragments

Details

Caches may be freely nested, as each cache is scoped to the template context it is executed in.

Caching occurs a small overhead for first-time usage, but is faster in proceeding calls.

Caching is powered by memoise::memoise. Cache storage limits, eviction, and persistence are controlled via the underlying memoise/cache backend.

If telemetry from otel is enabled, cache hit and miss events are recorded on the current active span (typically the route-level span made by routr). Hit and miss counts are also measured as metrics when enabled.

Note: invalidation affects future renders only. Calling this within the `cache()` block that is being targeted will not result in an invalidation of the cache.

See Also

template, set_cache_backend, get_cache_backend, api_cget, memoise::memoise

Examples

```
# Caching
nav <- template(user, {
  div(
    cache(
      "nav",
      vary = user$id,
      ttl = NULL,
      ul(
        li("Home"),
        li("Profile"),
        if (user$is_admin) li("Admin")
      )
    )
  )
})
nav(list(id = 1, is_admin = TRUE))

# Nested Caches
dashboard <- template(page = list(), stats = list(), recent = list(), {
    cache(
        name = "page",
        vary = page$updated_at,
        ttl = NULL,
        div(
            h1("Dashboard"),
            cache(
                name = "stats",
                vary = stats$updated_at,
                ttl = NULL,
                div(p(stats$count))
            ),
            cache(
                name = "recent",
```

```
                vary = recent$updated_at,
                ttl = NULL,
                div(recent)
            )
        )
    )
})
dashboard()

# TTL-caching (time-based invalidation)
page <- template({
  cache(
    name = "clock",
    vary = NULL,
    ttl = 60L,
    div(sprintf("Generated at %s", Sys.time()))
  )
})
page()

# Invalidate the current cache
# during rendering

page <- template(user, {
  if (user$refresh) {
    invalidate_cache_here(
      name = "content",
      vary = user$id
    )
  }
  div(
    cache(
      name = "content",
      vary = user$id,
      ttl = NULL,
      {
        p("Hello ", user$id)
      }
    )
  )
})

page(list(id = 1, refresh = FALSE))
page(list(id = 1, refresh = TRUE))
```

---

csrf_token *CSRF Token*

Description

csrf_token() returns the current CSRF token string for the active request when used within a template(). Calling it outside of a template() context will result in an error.

In most cases, CSRF tokens are inserted automatically for standard form helpers. Intended for custom forms / custom token placement (meta tags, JS fetch, etc).

Usage

```
csrf_token()
```

See Also

api_csrf

Examples

```
page <- template({
    html(
        head(
            meta(name = "csrf-token", content = csrf_token())
        ),
        body(
            div("App content")
        )
    )
})
page()
```

---

current_path *Get request data from current context*

Description

These read-only helpers provide access to request data for the current HTTP request via the freshwater request context.

- current_path() returns the request URL path
- current_method() returns the HTTP method
- current_query() returns the query parameters
- current_cookie() returns the value of a cookie by name
- current_header() returns the value of a header by name

These functions are primarily intended for use inside templates where a request context has been established. If called outside of an active context, an error is raised.

Context is available when freshwater context middleware is active (installed automatically by api_csrf(), api_error_pages(), or api_freshwater()).

Usage

```
current_path()

current_method()

current_query()

current_cookie(name)

current_header(name, normalise = TRUE)
```

Arguments

name	the name of a cookie or header
normalise	whether to normalise the provided name or pass it verbatim

See Also

api_freshwater(), api_csrf(), api_error_pages()

---

document *HTML Document Root*

Description

Constructs a full HTML document. It does not modify or validate its contents.

document() is used for full-page responses, and should not be used for partials, fragments or nested templates.

Usage

```
document(...)
```

Arguments

...	user-supplied content

Value

An htmltools::tagList, consisting of a doctype declaration, an tag, and user-supplied content.

See Also

template, fragment

Examples

```
document(
     htmltools::tags$head(
         htmltools::tags$title("Home")
     ),
     htmltools::tags$body(
         htmltools::tags$h1("Hello")
     )
)
```

`endpoints` *Reverse Routing*

Description

Access generated endpoint URL helpers.

Usage

```
endpoints(route = NULL, api = NULL, refresh = FALSE)
```

Arguments

`route`	the route group to retrieve endpoints from; typically defined either via the file name, routeName or route in plumber2. If NULL, will return all endpoints for all routes.
`api`	a plumber2::plumber2 api object. If NULL, context is used to find the api.
`refresh`	force refresh the registered routes. Useful if you have added routes after calling `endpoints()`.

Details

Alias rules:

- "/" endpoints become "index"
- GET endpoints are accessed directly, like index()
- non-GET endpoints require an accessor, like index$delete()
- path parameters are removed from the alias and used to disambiguate overloaded helpers via named function args
- Reserved argument: `.query` argument constructs a query from a named list
- Reserved argument: `.anchor` constructs an anchor from a character scalar

For example:

- `GET / -> index()`
- `POST / -> index$post()`

- GET /my/filter -> my_filter()
- GET /users/:id -> users(id = 1, .query = list(page = 2))
- GET /users/:id -> users(id = 1, .anchor = "details")
- GET /users/:id -> users(id = 1, .query = list(page = 2), .anchor = "details")
- DELETE /users/:id -> users$delete(id = 1)
- DELETE /users/:name -> users$delete(name = "Jim")

Ambiguous endpoint shapes will result in an error. It is recommended to ensure endpoints have different shapes. For example, the following cannot be disambiguated by endpoints() and thus results in an error:

- GET /foo/:id/bar -> foo_bar(id = _)
- GET /foo/bar/:id -> foo_bar(id = _)

Value

If route is NULL, returns a list of route groups and their endpoints. Otherwise returns a list of a route's endpoint accessors.

Examples

```
#* @plumber
function(api) {
   api |>
     api_freshwater()
}

#* @get /
#* @serializer html
#* @routeName user
function() {
     endpoints("user")$index()
}
```

---

form *Form*

Description

When used within a template(), a form implementation is injected that wraps htmltools::tags$form().

Usage

```
form(..., method = "get")
```

Arguments

`...`	tag attributes and children passed to the `htmltools::tags$form()` function
`method`	character scalar denoting the HTTP method to perform. One of: • "get" • "post" • "put" • "patch" • "delete"

Details

When a request context is available, freshwater adds optional behaviors such as CSRF token insertion and HTTP method spoofing.

Calling `form()` outside of `template()` rendering will result in an error. For a plain form tag in normal R code, use htmltools::tags`$form()`.

CSRF:

- If CSRF middleware is active, a hidden `csrf_token` input is automatically injected.

Method Spoofing:

If `method` is one of "put", "patch", or "delete", a hidden `_method` input is added and the HTML form method is set to "post".

Browsers only support GET and POST. When method is "put", "patch", or "delete", freshwater renders a POST form with a hidden _method field. Middleware interprets this as the effective HTTP method. Requires freshwater context-enabled middleware.

Value

(When injected) An `htmltools::tag` object.

See Also

template, api_csrf, api_freshwater, htmltools::tags

Examples

```
page <- template({
     form(method = "delete")
})
page()
```

freshwater_context *Freshwater Request Context*

Description

freshwater installs a per-request execution context that allows `current_path()`, `csrf_token()`, and `template()` helpers to access the active HTTP request. The context itself is stored in freshwater's internal state and is set/unset with each request. Context-dependent helpers are only valid when handling an active request. Moreover, requests are only active during *synchronous execution*.

Context is created automatically when `api_freshwater()`, `api_csrf()`, or `api_error_pages()` is installed.

Method spoofing is applied during the `before-request` phase by rewriting `REQUEST_METHOD` when a hidden `_method` field is present. This only applies to browser form submissions (i.e. Content-Type `application/x-www-form-urlencoded` or `multipart/form-data`).

Lifecycle

The context exists only during an active HTTP request. Calling context-dependent helpers outside a request will raise a `freshwater_context_missing` error.

See Also

`api_freshwater()`, `current_path()`

freshwater_error_templates *Error Page Templates*

Description

freshwater provides a number of default views that are served to HTML clients in the event of common HTTP error codes.

Usage

```
default_error_500_template(
  error = NULL,
  request = NULL,
  is_debug = FALSE,
  ...,
  fragment = NULL
)

default_error_404_template(error = NULL, request = NULL, ..., fragment = NULL)

default_error_403_template(error = NULL, request = NULL, ..., fragment = NULL)
```

Arguments

error	the error condition signaled by an error in the server's route handler
request	the reqres::Request request object the handler is responding to
is_debug	whether to provide the stack trace and error message to the web client. Although useful during development, it is heavily recommended to set as FALSE in production as it can leak sensitive information.
...	unused
fragment	unused

See Also

api_error_pages, template

---

get_cache_backend *Get freshwater's current cache backend*

Description

Returns the cache backend currently used by cache().

Usage

```
get_cache_backend()
```

Value

A cache backend object (typically from cachem), or NULL if the cache has not yet been initialised.

See Also

set_cache_backend(), cache()

Examples

```
get_cache_backend()
```

map_tags *Apply template function to each element of a vector*

Description

`map_tags()` returns a type-safe tag list, where each element is resolved by applying `.f` to each element of `.x`. If an element in `.x` is a falsey value (i.e. `NA`, `NaN`, `FALSE`, or `NULL`), the fallback value from `.empty` is used.

Additional arguments should be passed with an anonymous function.

Usage

```
map_tags(.x, .f, .empty = NULL)
```

Arguments

`.x`	list or atomic vector
`.f`	a function that takes a single argument returns a character vector, tag, or tagList.
`.empty`	fallback value for falsey elements

Details

Element values are evaluated prior to returning the tag list:

- NULL values are removed from the final tag list. The return length of `map_tags()` is therefore less than or equal to the length of `.x`
- All return values must be either a "shiny.tag", "shiny.tag.list", or "character" vector. An error is raised if an unexpected value is encountered.

See Also

template, base::lapply, htmltools::tagList

Examples

```
tpl <- template(x, {p(x)})
map_tags(seq(5L), tpl)

# falsey values are removed
map_tags(c(TRUE, FALSE, TRUE), tpl)
```

`redirect` *Redirect to another resource*

Description

If `after` is NULL, sends a 303 response and halts request processing. Client is redirected to the given location. This is commonly used in Post/Redirect/GET (PRG) setups to redirect clients to a new page following form submissions.

Usage

```
redirect(response, location, after = NULL, external = FALSE)
```

Arguments

`response`	reqres::Response object
`location`	path or url to redirect to
`after`	optional number of seconds to wait before redirection
`external`	whether to permit off-site redirects

Details

If after is a numeric, a "Refresh" header is attached to the response, instructing the browser to navigate to `location` after the specified number of seconds.

By default, absolute and cross-origin locations result in an error. If you wish to intentionally redirect outside the current origin, specify `external=TRUE`.

The delayed redirect uses the non-standard "Refresh" HTTP header which is widely supported by browsers but is not part of the official HTTP specification. It should not be relied on for API & non-browser clients.

See also:

- https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Redirections
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Refresh

Value

- plumber2::Break when issuing an immediate redirect.
- plumber2::Next when issuing a delayed navigation.

Examples

```
# Immediate redirect (PRG pattern)
#* @get /
function(response) {
    print("Hello!")
    redirect(response, "/foo")
}

# Delayed redirect after rendering content
#* @get /count/<n>
function(n, response) {
  redirect(response, "/", after = 1)
  paste("n =", n)
}
```

`register_async_evaluator` *Register context-safe async evaluator*

Description

Registers an asynchronous evaluator for routes, allowing freshwater context to be propagated to mirai::mirai workers. This means that contextful helpers such as current_method, current_path, and current_query (among others) will work in async routes.

Usage

```
register_async_evaluator(set_default = TRUE)
```

Arguments

`set_default`	whether to set the default async evaluator to `"freshwater"`

Details

Registration affects global plumber2 state, not just the current API process.

Context is not inherently portable across asynchronous request contexts, this function creates a portable snapshot of the current context that is passed to a mirai worker.

Hooks are *not* applied to the async route, but may be provided to any associated `then` handlers. If error pages are installed on the main process, errors from the worker will be appropriately converted into freshwater error pages. If CSRF protection is enabled, tokens will be propagated to the worker, ensuring async routes are still protected.

As `cache()` is process-local by default, memoised functions are *not* ported to workers. Likewise, `clear_cache()` and `invalidate_cache()` will only impact the local process' cache. If a shared cache is desired, consider configuring `cachem::cache_disk()` for caching, which will allow all process to utilise a shared cache. Note that TTL is process-local regardless of backend strategy used.

Requires the promises::promises, mirai::mirai, and mori::mori packages.

See Also

api_freshwater, api_error_pages, api_hooks, mirai::mirai, current_method

Examples

```
register_async_evaluator()
#* @async
#* @get /async
function() {
 current_path()
}
```

`register_html_serialiser` *Register HTML Serialiser*

Description

Registers an HTML serialiser for plumber2::plumber2 that renders shiny tags and taglists via `htmltools::doRenderTags()`. This is preferrable over `htmltools::renderTags()` as often we want to be able to emit head tags which htmltools::htmltools attempts to hide for shiny usecases.

Usage

```
register_html_serialiser(force = FALSE)
```

Arguments

`force`	bypass checks and re-register the freshwater serialiser

Details

The freshwater serialiser safely falls back to the default plumber2 implementation for other classes of inputs.

Specifically, if an input inherits `shiny.tag`, `shiny.tag.list` or `html`, it will be serialised via `htmltools::as.tags()` and `htmltools::doRenderTags()`. Otherwise, the default HTML serialiser will be used. If missing, the input will be coerced via `as.character()`.

Registration affects global process state, not just the current API process.

set_cache_backend *Configure freshwater's cache backend*

Description

`set_cache_backend()` replaces the cache backend used by `cache()`.

Usage

```
set_cache_backend(backend)
```

Arguments

backend	cache backend accepted by memoise::memoise

Details

This function allows controlling cache persistence (memory vs disk), eviction policies, and storage limits via the backend object.

See Also

`cache()`, `clear_cache()`, memoise::memoise

target *Resolve a template's target selector*

Description

`target()` returns a CSS selector for a template instance. For normal targets, this is an id selector of the form `[id="<TEMPLATE_ID>"]`, where the id is resolved from the template's id function. Root target assumes single-root output. Use `.part` for multi-root templates.

Usage

```
target(tpl, ..., .part = NULL)
```

Arguments

tpl	a freshwater template
...	arguments passed to the template id function
.part	whether to select a sub-part

Details

In order to support multi-root template selection, supplying `.part` will return a data attribute selector of the form: `[data-fw-part="<TEMPLATE_ID>-<PART_NAME>"]`

Part names are automatically scoped against the template's `.id`, ensuring unique data attributes across templates.

See Also

targets, template

Examples

```
card <- template(
    user,
    .id = function(user) sprintf("user-%s", user$id),
    {
        div(user$name)
    }
)
target(card, list(id = 1234L))

user_table <- template(users = list(), .id = "my-table", {
    table(
        thead(
            .part = "header",
            tr(th("Name"))
        ),
        tbody(
            .part = "body",
            lapply(users, \(user) tr(td(user)))
        ),
        tfoot(
            .part = "footer",
            tr(td(sprintf("There are '%s' users.", length(users))))
        )
   )
})
target(user_table, .part = "body")
```

`targets` *Combine multiple target selectors*

Description

`targets()` returns a comma-separated string combining multiple `target()` calls. If a template is supplied, `target()` is called on it, otherwise the value is coerced to character and used as-is.

Usage

```
targets(...)
```

Arguments

...	templates or character vectors

See Also

target

Examples

```
tpl <- template(.id = "foo", {})
tpl2 <- template(x, .id = function(x) x, {})
targets(
    tpl,
   target(tpl2, x = "bar")
)
```

---

template *Create a reusable HTML template*

Description

`template()` is a function factory that captures a template expression and returns a callable HTML renderer. The expression is evaluated under the `htmltools::withTags()` environment, so tag functions such as `div()` or `p()` are available.

Templates may define:

- **parameters**: symbols or named defaults which are used as arguments to the renderer
- **content injection**: if the template uses `...`, the renderer passes `...` to the containing HTML nodes defined in the template.
- **fragments**: named subtemplates that can be optionally extracted from the template upon rendering by supplying `fragment = "name"`. Fragment names are required. If multiple fragment names are specified, fragments will be extracted and collated into a htmltools::tagList, in the order of names provided. If a specified fragment cannot be found, an error will be raised.

Usage

```
template(..., .id = NULL, .envir = rlang::caller_env())

fragment(name = NULL, ...)
```

Arguments

...	template definition. Provide zero or more parameters, followed by a single braced expression.
.id	a character scalar or function that returns a character scalar. The result is provided as an id attribute to the root of the template.
.envir	the environment in which to evaluate the template
name	the name of the fragment

Value

function of class `template` with interface `fn(<declared params>, ..., fragment = NULL)`

Attributes

Attributes with non-leading underscores are rewritten as hyphenated versions instead. This means you can write `htmltools::div(data_foo="bar")` which is converted to `htmltools::div(data-foo="bar")`.

Attributes with trailing underscores have their underscores stripped. This means the you can write `htmltools::tags$label(for_="foo")` which is converted to `htmltools::tags$label(for="foo")`.

An escape hatch exists If you explicitly want underscores in your attributes. You may use double underscores, which will be converted to single underscores e.g. `htmltools::div(data__foo="bar")` which is converted to `htmltools::div(data_foo="bar")`.

Template IDs

Templates can define an `.id` string, or a function that resolves a stable HTML `id` based on parameters passed to the template. Default template arguments are provided to the id. See `target()` for more information.

Built-in Helpers

Template bodies are evaluated in a freshwater environment that provides the following helpers:

- `form()` — form helper with optional CSRF injection and method spoofing.
- `csrf_token()` — returns the current CSRF token string

Template Context

A template render context is maintained during evaluation which is used for fragment extraction and cache scoping. The template context is separate from the request context defined elsewhere.

See Also

document, cache, form, target, csrf_token, api_freshwater

Examples

```
# Example Fragment Usage
page_main <- template(
    {
        div(
            h1("Dashboard"),
            fragment(
                 p("Welcome back"),
                 name = "content"
            ),
            small("2026")
        )
    }
)
```

```
page_main(fragment="content")

# Template slots

details <- template(name, age, {
    nm <- sprintf("Hello, my name is: %s", name)
    old <- sprintf("I am %s years old.", age)
    div(
        p(nm),
        p(old)
    )
})

details("Jim", 30)

# Templates and fragments can also be combined

card <- template(
    ttl, footer = NULL, {
        div(
            h2(ttl),
            fragment(div("Card body"), name="body"),
            if (!is.null(footer)) {
                fragment(
                    div(footer),
                    name = "footer"
                )
            }
        )
    }
)
card("Card Title")
card("Card Title", fragment="body")
card("Card Title", "Footer text", fragment = "footer")

# Dots (content injection)
layout <- template({
    htmltools::tagList(
        head(meta(title = "foo")),
        body(...)
    )
})

layout(htmltools::div("content"))

# Attribute Norming
my_form <- template({
    form(
        label(for_ = "desc", "Description"),
        input(
            type = "text",
            id = "desc",
```

```
                data_user__id = "123"
            )
        )
    })
    my_form()
```

Index