Package 'routr'

May 9, 2026

Type Package

Title A Simple Router for HTTP and WebSocket Requests

Version 2.0.0

Maintainer Thomas Lin Pedersen <thomasp85@gmail.com>

Description In order to make sure that web request ends up in the correct handler function a router is often used. 'routr' is a package implementing a simple but powerful routing functionality for R based servers. It is a fully functional 'fiery' plugin, but can also be used with other 'httpuv' based servers.

License MIT + file LICENSE

Encoding UTF-8

Imports R6, reqres (>= 1.2.0), stringi, rlang (>= 1.1.0), cli, lifecycle, fs, promises (>= 1.5.0), brio, otel, waysign, glue

RoxygenNote 7.3.3

Suggests testthat (>= 3.0.0), covr, fiery (>= 1.3.0), swagger, redoc, rapidoc, rmarkdown, quarto, mirai, knitr, later, jsonlite, yaml

URL https://routr.data-imaginist.com, https://github.com/thomasp85/routr

BugReports https://github.com/thomasp85/routr/issues

Config/testthat/edition 3

NeedsCompilation no

Author Thomas Lin Pedersen [cre, aut] (ORCID: <https://orcid.org/0000-0002-5147-4711>)

Depends R (>= 4.1.0)

Repository CRAN

Date/Publication 2025-12-18 12:20:02 UTC

Contents

`AssetRoute` *Static file serving*

Description

A class for serving files from the server directly. The `AssetRoute` is fundamentally different than the other routes provided by routr. It is specific to httpuv and circumvents the standard dispatch entirely (the request never enters the R process). This makes it extremely fast but also somewhat limited as you can't pass the request through any middleware.

Active bindings

`at` The url path to serve the assets on

`path` The path to the file or directory to serve

`use_index` Should an `index.html` file be served if present when a client requests the folder

`fallthrough` Should requests that doesn't match a file enter the request loop or have a 404 response send directly

`html_charset` The charset to report when serving html files

`headers` A list of headers to add to the response.

`validation` An optional validation pattern to compare to the request headers

`except` One or more url paths that should be excluded from this route

`name` An autogenerated name for the asset route

Methods

Public methods:

- AssetRoute$new()
- AssetRoute$print()
- AssetRoute$on_attach()
- AssetRoute$clone()

Method `new()`**:** Create a new AssetRoute

Usage:

```
AssetRoute$new(
  at,
  path,
  use_index = TRUE,
  fallthrough = FALSE,
  html_charset = "utf-8",
  headers = list(),
  validation = NULL,
  except = NULL
)
```

Arguments:

`at` The url path to listen to requests on

`path` The path to the file or directory on the file system

`use_index` Should an `index.html` file be served if present when a client requests the folder

`fallthrough` Should requests that doesn't match a file enter the request loop or have a 404 response send directly

`html_charset` The charset to report when serving html files

`headers` A list of headers to add to the response. Will be combined with the global headers of the app

`validation` A string for validating incoming requests. See httpuv::staticPath

`except` One or more url paths that should be excluded from the route. Requests matching these will enter the standard router dispatch. The paths are interpreted as subpaths to `at`, e.g. the final path to exclude will be `at+exclude`

Method `print()`**:** Pretty printing of the object

Usage:

```
AssetRoute$print(...)
```

Arguments:

`...` Ignored

Method `on_attach()`**:** Method for use by `fiery` when attached as a plugin. Should not be called directly. This method creates a RouteStack with the asset route as the single route and then mounts that to the app. For more flexibility create the RouteStack manually

Usage:

```
AssetRoute$on_attach(app, on_error = deprecated(), ...)
```

Arguments:

`app` The Fire object to attach the router to

`on_error` **[Deprecated]** A function for error handling

`...` Ignored

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
AssetRoute$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`asset_route` *High performance route for serving static files*

Description

An `asset_route()` is fundamentally different than the other routes provided by routr. Conceptually it is akin to `resource_route()` in that it is used for serving static file content, but this route circumvents the standard dispatch entirely (the request never enters the R process). This makes it extremely fast but also somewhat limited as you can't pass the request through any middleware. The choice between `asset_route()` and `resource_route()` thus depends on your needs.

Usage

```
asset_route(
  at,
  path,
  use_index = TRUE,
  fallthrough = FALSE,
  html_charset = "utf-8",
  headers = list(),
  validation = NULL,
  except = NULL
)
```

Arguments

`at`	The url path to listen to requests on
`path`	The path to the file or directory on the file system
`use_index`	Should an `index.html` file be served if present when a client requests the folder
`fallthrough`	Should requests that doesn't match a file enter the request loop or have a 404 response send directly
`html_charset`	The charset to report when serving html files
`headers`	A list of headers to add to the response. Will be combined with the global headers of the app

validation	An optional validation pattern. Presently, the only type of validation supported is an exact string match of a header. For example, if validation is '"abc" = "xyz"', then HTTP requests must have a header named abc (case-insensitive) with the value xyz (case-sensitive). If a request does not have a matching header, than httpuv will give a 403 Forbidden response. If the character(0) (the default), then no validation check will be performed.
except	One or more url paths that should be excluded from the route. Requests matching these will enter the standard router dispatch. The paths are interpreted as subpaths to at, e.g. the final path to exclude will be at+exclude (see example)

Value

An AssetRoute object

See Also

Other Route constructors: openapi_route(), resource_route(), shared_secret_route(), sizelimit_route()

Examples

```
asset_route("/wd", "./", except = "/private")
```

---

openapi_route *Create a route for serving OpenAPI documentation of your server*

Description

This route facilitates serving the OpenAPI specs for your server, using either RapiDoc, Redoc or Swagger as a UI for it. This function does not help you describe your API - you have to provide the description for it yourself.

Usage

```
openapi_route(
  spec,
  root = "__docs__",
  ui = c("rapidoc", "redoc", "swagger"),
  ...
)
```

Arguments

spec	The path to the json or yaml file describing your OpenAPI spec
root	The point from which you want to serve your UI from
ui	Either "rapidoc", "redoc" or "swagger", setting which UI to use
...	Further arguments passed on to the ui functions (e.g. rapidoc::rapidoc_spec())

Value

A Route object

See Also

Other Route constructors: `asset_route()`, `resource_route()`, `shared_secret_route()`, `sizelimit_route()`

`report_route` *Create a route that renders and serves an Rmarkdown or Quarto report*

Description

This route allows you to serve a report written as a Quarto/Rmarkdown document. The report will be rendered on demand using the query params as parameters for the report if they match, or by providing them in the body of a POST request. Depending on the value of the value of `max_age` the rendered report is kept and served without a re-render on subsequent requests. The rendering can happen asynchronously in which case a promise is returned.

Usage

```
report_route(
  path,
  file,
  ...,
  max_age = Inf,
  async = TRUE,
  finalize = NULL,
  continue = FALSE,
  ignore_trailing_slash = FALSE,
  cache_dir = tempfile(pattern = "routr_report"),
  cache_by_id = FALSE,
  param_caster = identity
)
```

Arguments

`path`	The url path to serve the report from
`file`	The quarto or rmarkdown file to use for rendering of the report
`...`	Further arguments to `quarto::quarto_render()` or `rmarkdown::render()`
`max_age`	The maximum age in seconds to keep a rendered report before initiating a re-render
`async`	Should rendering happen asynchronously (using mirai)
`finalize`	An optional function to run before sending the response back. The function will receive the request as the first argument, the response as the second, and the server as the third.

`continue` A logical that defines whether the response is returned directly after rendering or should be made available to subsequent routes

`ignore_trailing_slash` Should `path` be taken exactly or should both a version with and without a terminating slash be accepted

`cache_dir` The location of the render cache. By default a temporary folder is created for it.

`cache_by_id` Should caching be scoped by the user id. If the rendering is dependent on user-level access to different data this is necessary to avoid data leakage.

`param_caster` An optional function to convert the query/body parameters into the expected type, or a list with elements `query` and `body` each holding a function to convert their respective parts into the expected type.

Details

Only the formats explicitely stated in the header of the report are allowed and they can be selected in multiple ways. Either by appending the name of the format as a subpath to the path (e.g. `/report/revealjs`), by appending the extension of the output type to the path (e.g. `/report.pdf`), or by standard content negotiation using the `Content-Type` header of the request. For the latter two, it is only possible to select the first format of any kind that has the same mime-type/extension.

Value

A route object

`resource_route` *Create a route for fetching files*

Description

This function creates a route mapping different paths to files on the server filesystem. Different subpaths can be mapped to different locations on the server so that e.g. `/data/` maps to `/path/to/data/` and `/assets/` maps to `/a/completely/different/path/`. The route support automatic expansion of paths to a default extension or file, using compressed versions of files if the request permits it, and setting the correct headers so that results are cached.

Usage

```
resource_route(
  ...,
  default_file = "index.html",
  default_ext = "html",
  finalize = NULL,
  continue = FALSE
)
```

Arguments

...	Named arguments mapping a subpath in the URL to a location on the file system. These mappings will be checked in sequence
default_file	The default file to look for if the path does not map to a file directly (see Details)
default_ext	The default file extension to add to the file if a file cannot be found at the provided path and the path does not have an extension (see Details)
finalize	An optional function to run if a file is found. The function will receive the request as the first argument, the response as the second, and anything passed on through ... in the dispatch method. Any return value from the function is discarded. The function must accept ...
continue	A logical that should be returned if a file is found. Defaults to FALSE indicating that the response should be send unmodified.

Details

The way paths are resolved to a file is, for every mounted location,

1. Check if the path contains the mount point. If not, continue to the next mount point
2. substitute the mount point for the local location in the path
3. if the path ends with / add the default_file (defaults to index.html)
4. see if the file exists along with compressed versions (versions with .gz, .zip, .br, .zz appended)
5. if any version exists, chose the prefered encoding based on the Accept-Encoding header in the request, and return.
6. if none exists and the path does not specify a file extension, add default_ext to the path and repeat 3-4
7. if none exists still and the path does not specify a file extension, add default_file to the path and repeat 3-4
8. if none exists still, continue to the next mount point

This means that for the path /data/mtcars, the following locations will be tested (assuming the /data/ -> /path/to/data/ mapping):

1. /path/to/data/mtcars, /path/to/data/mtcars.gz, /path/to/data/mtcars.zip, /path/to/data/mtcars.br, /path/to/data/mtcars.zz
2. /path/to/data/mtcars.html, /path/to/data/mtcars.html.gz, /path/to/data/mtcars.html.zip, /path/to/data/mtcars.html.br, /path/to/data/mtcars.html.zz
3. /path/to/data/mtcars/index.html, /path/to/data/mtcars/index.html.gz, /path/to/data/mtcars/index.h /path/to/data/mtcars/index.html.br, /path/to/data/mtcars/index.html.zz

Assuming the default values of default_file and default_ext

If a file is not found, the route will simply return TRUE to hand of control to subsequent routes in the stack, otherwise it will return the logical value in the continue argument (defaults to FALSE, thus shortcutting any additional routes in the stack).

If a file is found the request headers If-Modified-Since and If-None-Match, will be fetched and, if exist, will be used to determine whether a 304 - Not Modified response should be send instead

of the file. If the file should be send, it will be added to the response along with the following headers:

- `Content-Type` based on the extension of the file (without any encoding extensions)
- `Content-Encoding` based on the negotiated file encoding
- `ETag` based on `rlang::hash()` of the last modified date
- `Cache-Control` set to `max-age=3600`

Furthermore `Content-Length` will be set automatically by `httpuv`

Lastly, if found, the finalize function will be called, forwarding the `request`, `response` and `...` from the `dispatch` method.

Value

A Route object

See Also

Other Route constructors: `asset_route()`, `openapi_route()`, `shared_secret_route()`, `sizelimit_route()`

Examples

```
# Map package files
res_route <- resource_route(
  '/package_files/' = system.file(package = 'routr')
)

rook <- fiery::fake_request('http://example.com/package_files/DESCRIPTION')
req <- reqres::Request$new(rook)
res_route$dispatch(req)
req$response$as_list()
```

Route *Single route dispatch*

Description

The `Route` class is used to encapsulate a single URL dispatch, that is, chose a single handler from a range based on a URL path. A handler will be called with a request, response, and keys argument as well as any additional arguments passed on to `dispatch()`.

Method matching

A handler is referencing a specific HTTP method (`get`, `post`, etc.) but can also reference `all` to indicate that it should match all types of requests. Handlers referencing `all` have lower precedence than those referencing specific methods, so will only be called if a match is not found within the handlers of the specific method.

Path matching

The path will be stripped the query string prior to handler lookup. routr is using the waysign package to match URL paths to the path pattern provided along with the handler. A path pattern consists of zero or more elements separated by `/`, each element can be one of three basic types:

- **Fixed:** Is a string literal that will be matched exactly. The pattern `/user/thomas` consists of two fixed elements and will only ever be matched exactly to `/user/thomas`
- **Parameterized:** Is a variable that can take the value of any string (not including a `/`). A parameter consist of a `:` followed by a name (made up of alphanumeric characters). The patter `/user/:id` consist of a literal and a parameter and will match to e.g. `/user/thomas` and `user/hana`, but not `user/thomas/settings`. A parameter doesn't have to take up all of an element, it can be a mix of literal and one or more parameters, e.g. `/posts/date-:year-:month-:day` will match to `posts/date-2025-11-05`. If you want to add an alphanumeric literal end to a parameterized element you can separate it by `\\` like `/posts/:title\\post` which will match `/posts/hello_worldpost`. A parameter can be made optional by terminating it with `?`. `/user/:id?` will match both `/user/thomas` and `/user/`. While optional parameters are most useful in the end of a path they can also be in the middle of a pattern, e.g. `/user/:id?/settings` which will match `/user/thomas/settings` and `/user//settings` (note the double slashes)
- **Wildcards:** Is like parameters except they can take up multiple elements (i.e. the match to strings that contain `/`). The come in two flavors: one-or-more and zero-or-more. The first uses a `+` and the latter a `*`. These can and should be named be prepending a parameter name to the operator (e.g. `:name+`). The pattern `/user/:id+` will match `/user/thomas` and `user/thomas/settings`, the pattern `user/:id*` will match those two as well and additionally match `/user/`.

The syntax allows for multiple patterns matching to the same string, e.g. `/posts/:date`, `/posts/:day-:month-:year`, and `/posts/:remainder+` all matches to `/posts/03-09-2024`. waysign resolves this by always matching to the most specific pattern. Literals are more specific than parameters which are more specific than wildcards. Further, a element consisting of multiple parameters are considered more specific than one consisting of fewer.

Handler calling

Handlers are only called for their side-effects and are expected to return either `TRUE` or `FALSE` indicating whether additional routes in a `RouteStack` should be called, e.g. if a handler is returning `FALSE` all further processing of the request will be terminated and the response will be passed along in its current state. Thus, the intend of the handlers is to modify the request and response objects, in place.

When the handler is called it will be passed in a request object to the `request` argument, a response object to the `response` argument and a list to the `keys` argument. The names of the elements in the `keys` list will match those given in the pattern (excluding the `:`) and the value will be the part it matched to. If wildcards are unnamed they will be named after their index and type, e.g. `/path/+/and/some/more/*` will automatically name the two wildcards `+1` and `*2`. To avoid ambiguity and errors it is recommended to explicitly name wildcards if you intend to use their value for anything. In addition to `request`, `response`, and `keys` any argument passed to the `...` in the `dispatch()` method is also passed into the handler.

Initialization

A new 'Route'-object is initialized using the `new()` method on the generator or alternatively by using `route()`:

Usage

```
route <- Route$new(...)
route <- route(...)
```

Active bindings

`root` The root of the route. Will be removed from the path of any request before matching a handler

`name` An autogenerated name for the route

`empty` Is the route empty

Methods

Public methods:

- `Route$new()`
- `Route$print()`
- `Route$add_handler()`
- `Route$remove_handler()`
- `Route$get_handler()`
- `Route$remap_handlers()`
- `Route$merge_route()`
- `Route$dispatch()`
- `Route$on_attach()`
- `Route$clone()`

Method `new()`**:** Create a new Route

Usage:

```
Route$new(..., ignore_trailing_slash = FALSE)
```

Arguments:

`...` Handlers to add up front. Must be in the form of named lists where the names corresponds to paths and the elements are the handlers. The name of the argument itself defines the method to listen on (see examples)

`ignore_trailing_slash` Logical. Should the trailing slash of a path be ignored when adding handlers and handling requests. Setting this will not change the request or the path associated with but just ensure that both `path/to/resource` and `path/to/resource/` ends up in the same handler. Because the request is left untouched, setting this to `TRUE` will not affect further processing by other routes

Method `print()`**:** Pretty printing of the object

Usage:

```
Route$print(...)
```

Arguments:

`...` Ignored

Method `add_handler()`**:** Add a handler to the specified method and path. The special method `'all'` will allow the handler to match all http request methods. The path is a URL path consisting of strings, parameters (strings prefixed with `:`), and wildcards (`*`), separated by `/`. A wildcard will match anything and is thus not restricted to a single path element (i.e. it will span multiple `/` if possible). The handler must be a function containing the arguments `request`, `response`, `keys`, and `...`, and must return either `TRUE` or `FALSE`. The `request` argument will be a reqres::Request object and the `response` argument will be a reqres::Response object matching the current exchange. The `keys` argument will be a named list with the value of all matched parameters from the path. Any additional argument passed on to the `dispatch` method will be avaiable as well. This method will override an existing handler with the same method and path.

Usage:

```
Route$add_handler(method, path, handler, reject_missing_methods = FALSE)
```

Arguments:

`method` The http method to match the handler to

`path` The URL path to match to

`handler` A handler function

`reject_missing_methods` Should requests to this path that doesn't have a handler for the specific method automatically be rejected with a 405 Method Not Allowed response with the correct Allow header informing the client of the implemented methods. Assigning a handler to `"all"` for the same path at a later point will overwrite this functionality. Be aware that setting this to `TRUE` will prevent the request from falling through to other routes that might have a matching method and path.

Method `remove_handler()`**:** Removes the handler assigned to the specified method and path. If no handler have been assigned it will silently ignore it.

Usage:

```
Route$remove_handler(method, path)
```

Arguments:

`method` The http method of the handler to remove

`path` The URL path of the handler to remove

Method `get_handler()`**:** Returns a handler already assigned to the specified method and path. If no handler have been assigned it will return NULL.

Usage:

```
Route$get_handler(method, path)
```

Arguments:

`method` The http method of the handler to find

`path` The URL path of the handler to find

Method `remap_handlers()`**:** Allows you to loop through all added handlers and reassings them at will. A function with the parameters `method`, `path`, and `handler` must be provided which is responsible for reassigning the handler given in the arguments. If the function does not reassign the handler, then the handler is removed.

Usage:

```
Route$remap_handlers(.f)
```

Arguments:

`.f` A function performing the remapping of each handler

Method `merge_route()`**:** Merge another route into this one, adopting all its handlers. The other route will be empty after the merge.

Usage:

```
Route$merge_route(route, use_root = TRUE)
```

Arguments:

`route` A Route object

`use_root` Should the root of `route` be prepended to all paths from the route before adding them

Method `dispatch()`**:** Based on a reqres::Request object the route will find the correct handler and call it with the correct arguments. Anything passed in with `...` will be passed along to the handler.

Usage:

```
Route$dispatch(request, ..., .require_bool_output = TRUE)
```

Arguments:

`request` The request to route

`...` Additional arguments to the handlers

`.require_bool_output` Should the dispatch enforce a boolean output. Mainly for internal use.

Method `on_attach()`**:** Method for use by `fiery` when attached as a plugin. Should not be called directly. This method creates a RouteStack with the route as the single route and then mounts that to the app. For more flexibility create the RouteStack manually

Usage:

```
Route$on_attach(app, on_error = deprecated(), ...)
```

Arguments:

`app` The Fire object to attach the router to

`on_error` **[Deprecated]** A function for error handling

`...` Ignored

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Route$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

RouteStack for binding multiple routes sequentially

Examples

```
# Initialise an empty route
route <- Route$new()

# Initialise a route with handlers assigned
route <- Route$new(
  all = list(
    '/*' = function(request, response, keys, ...) {
      message('Request received')
      TRUE
    }
  )
)

# Remove it again
route$remove_handler('all', '/*')
```

`route` *Construct a new route*

Description

This function constructs a new Route, optionally with a set of handlers already attached.

Usage

```
route(..., root = "")
```

Arguments

`...`	Handlers to add up front. Must be in the form of named lists where the names corresponds to paths and the elements are the handlers. The name of the argument itself defines the method to listen on (see examples)
`root`	The root of the route. Will be removed from the path of any request before matching a handler

Value

A Route object

Examples

```
# An empty route
route <- route()
route

# Prepopulating it at construction
route <- route(all = list(
```

```
  '/*' = function(request, response, keys, ...) {
    message('Request received')
    TRUE
  }
))
route
```

RouteStack *Combine multiple routes for sequential routing*

Description

Combine multiple routes for sequential routing

Combine multiple routes for sequential routing

Details

The RouteStack class encapsulate multiple Routes and lets a request be passed through each sequentially. If a route is returning FALSE upon dispatch further dispatching is cancelled.

Initialization

A new 'RouteStack'-object is initialized using the new() method on the generator:

Usage

```
router <- RouteStack$new(..., path_extractor = function(msg, bin) '/')
```

Fiery plugin

A RouteStack object is a valid fiery plugin and can thus be passed in to the attach() method of a Fire object. When used as a fiery plugin it is important to be concious for what event it is attached to. By default it will be attached to the request event and thus be used to handle HTTP request messaging. An alternative is to attach it to the header event that is fired when all headers have been received but before the body is. This allows you to short-circuit request handling and e.g. reject requests above a certain size. When the router is attached to the header event any handler returning FALSE will signal that further handling of the request should be stopped and the response in its current form should be returned without fetching the request body.

One last possibility is to attach it to the message event and thus use it to handle WebSocket messages. This use case is a bit different from that of request and header. As routr uses Request objects as a vessel between routes and WebSocket messages are not HTTP requests, some modification is needed. The way routr achieves this is be modifying the HTTP request that established the WebSocket connection and send this through the routes. Using the path_extractor function provided in the RouteStack constructor it will extract a path to dispatch on and assign it to the request. Furthermore it assigns the message to the body of the request and sets the Content-Type header

based on whether the message is binary `application/octet-stream` or not `text/plain`. As WebSocket communication is asynchronous the response is ignored when attached to the `message` event. If communication should be send back, use `server$send()` inside the handler(s).

How a `RouteStack` is attached is defined by the `attach_to` field which must be either `'request'`, `'header'`, or `'message'`.

When attaching the `RouteStack` it is possible to modify how errors are handled, using the `on_error` argument, which will change the error handler set on the `RouteStack`. By default the error handler will be changed to using the `fiery` logging system if the `Fire` object supports it.

Active bindings

`attach_to` The event this routr should respond to

`name` An autogenerated name for the route stack

`routes` Gices the name of all routes in the stack

`empty` Is the route stack empty

Methods

Public methods:

- `RouteStack$new()`
- `RouteStack$print()`
- `RouteStack$add_route()`
- `RouteStack$add_redirect()`
- `RouteStack$get_route()`
- `RouteStack$has_route()`
- `RouteStack$remove_route()`
- `RouteStack$dispatch()`
- `RouteStack$dispatch_to_first_match()`
- `RouteStack$on_attach()`
- `RouteStack$merge_stack()`
- `RouteStack$clone()`

Method `new()`**:** Create a new RouteStack

Usage:

```
RouteStack$new(..., path_extractor = function(msg, bin) "/")
```

Arguments:

`...` Routes to add up front. Must be in the form of named arguments containing `Route` objects.

`path_extractor` A function that returns a path to dispatch on from a WebSocket message. Will only be used if `attach_to == 'message'`. Defaults to a function returning `'/'`

Method `print()`**:** Pretty printing of the object

Usage:

```
RouteStack$print(...)
```

Arguments:

... Ignored

Method `add_route()`**:** Adds a new route to the stack. `route` must be a `Route` object, `name` must be a string. If `after` is given the route will be inserted after the given index, if not (or `NULL`) it will be inserted in the end of the stack.

Usage:

```
RouteStack$add_route(route, name, after = NULL)
```

Arguments:

`route` A `Route` object

`name` The name of the route

`after` The location in the stack to put the route

Method `add_redirect()`**:** Adds a permanent (308) or temporary (307) redirect from a path to another. The paths can contain path arguments and wildcards, but all those present in `to` must also be present in `from` (the reverse is not required)

Usage:

```
RouteStack$add_redirect(method, from, to, permanent = TRUE)
```

Arguments:

`method` The http method to match the handler to

`from` The path the redirect should respond to

`to` The path the redirect should signal to the client as the new path

`permanent` Logical. If `TRUE` then a 308 Permanent Redirect is send back, instructing the client to update the URL in the browser to show the new path as well as avoid sending requests to the old URL again. If `FALSE` then a 307 Temporary Redirect is send back, instructing the client to proceed as if the response comes from the old path

Method `get_route()`**:** Get the route with a given name

Usage:

```
RouteStack$get_route(name)
```

Arguments:

`name` The name of the route to retrieve

Method `has_route()`**:** Test if the routestack contains a route with the given name.

Usage:

```
RouteStack$has_route(name)
```

Arguments:

`name` The name of the route to look for

Method `remove_route()`**:** Removes the route with the given name from the stack.

Usage:

```
RouteStack$remove_route(name)
```

Arguments:

`name` The name of the route to remove

Method `dispatch()`**:** Passes a reqres::Request through the stack of routes in sequence until one of the routes return `FALSE` or every route have been passed through. `...` will be passed on to the dispatch of each `Route` on the stack.

Usage:

```
RouteStack$dispatch(request, ...)
```

Arguments:

`request` The request to route

`...` Additional arguments to pass on to the handlers

Method `dispatch_to_first_match()`**:** asses a reqres::Request through the stack of routes in sequence until a handler is found. `...` will be passed on to the dispatch of each `Route` on the stack. This dispatch does not require the handler to return a boolean, and will return the value of the handler call or `NULL` if no handler is matched

Usage:

```
RouteStack$dispatch_to_first_match(request, ...)
```

Arguments:

`request` The request to route

`...` Additional arguments to pass on to the handlers

Method `on_attach()`**:** Method for use by `fiery` when attached as a plugin. Should not be called directly.

Usage:

```
RouteStack$on_attach(app, on_error = deprecated(), ...)
```

Arguments:

`app` The Fire object to attach the router to

`on_error` **[Deprecated]** A function for error handling

`...` Ignored

Method `merge_stack()`**:** Merge two route stacks together adding all routes from the other route to this. The other route stack will be empty after this.

Usage:

```
RouteStack$merge_stack(stack)
```

Arguments:

`stack` Another RouteStack object to merge into this one

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
RouteStack$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

See Also

Route for defining single routes

Examples

```
# Create a new stack
routes <- RouteStack$new()

# Populate it wih routes
first <- Route$new()
first$add_handler('all', '*', function(request, response, keys, ...) {
  message('This will always get called first')
  TRUE
})
second <- Route$new()
second$add_handler('get', '/demo/', function(request, response, keys, ...) {
  message('This will get called next if the request asks for /demo/')
  TRUE
})
routes$add_route(first, 'first')
routes$add_route(second, 'second')

# Send a request through
rook <- fiery::fake_request('http://example.com/demo/', method = 'get')
req <- reqres::Request$new(rook)
routes$dispatch(req)
```

`route_add` *Route handlers*

Description

These functions helps you to add, remove, and retrieve route handlers. They all (except for `route_get()`) returns the route for easy chaining.

Usage

```
route_add(x, method, path, handler)

route_remove(x, method, path)

route_get(x, method, path)
```

Arguments

`x`	A Route object
`method`	A request method for the handler. The special method `"all"` will allow the handler to match all http request methods that comes in.
`path`	A URL path for the handler. See Paths for more on path semantics
`handler`	A handler function. See Handlers for more about the semantics of handlers

Value

`x`, modified. `route_get()` returns the requested handler

Paths

The path is a URL path consisting of strings, parameters (strings prefixed with `:`), and wildcards (`*`), separated by `/`. A wildcard will match anything and is thus not restricted to a single path element (i.e. it will span multiple `/` if possible). When serving a request only a single handler is selected based on the path match that is most specific. Specificity is based on number of path parts (ie. number of elements separated by `/`), the more the better; number of wildcards, the fewer the better; and number of keys, the fewer the better. When printing the route you can see the priority of all the paths in the route as they are sorted by that

Handlers

The handler is a function. At the very least it should have a `...` argument and it must return eiter `TRUE` or `FALSE`. Returning `TRUE` means that the request is allowed to continue processing and can be passed on to the next route in the stack. Returning `FALSE` stops the processing of the request by the stack.

While any arguments besides `...` are optional, there are a few that will get passed in named:

- `request` will hold the request as a reqres::Request object
- `response` will hold the request as a reqres::Response object
- `keys` will be a named list containing the values of the matched path keys (see example)

Further, if routr is used as a fiery plugin, the handler will receive:

- `server` is the `fiery::Fire` object defining the app
- `id` is the id of the client sending the request, as provided by fiery
- `arg_list` is a list of values as calculated by the servers `before-request` event handlers

Any and all of the above can be ignored by your handler, but accepting the `server` is often paramount to more powerful features such as delayed execution or logging.

Examples

```
# Add a handler
route <- route() |>
  route_add("get", "/:what", function(request, response, keys, ...) {
    message("Requesting", keys$what)
    TRUE
  })
route

# Retrieve the handler
route |> route_get("get", "/:what")

# Remove the handler
route |> route_remove("get", "/:what")
```

route_merge *Merge one route into another*

Description

This function allows you to combine two separate routes into one. This is different from combining them in a routestack, because a request is only matched to one handler in each route (thus combining them with `route_merge()` will ensure only one handler is called).

Usage

```
route_merge(x, route, use_root = TRUE)
```

Arguments

x, route	Route objects to merge. `route` will be merged into `x`
use_root	Should the root of `route` be added to all its paths before it is merged into `x`

Value

`x` with `route` merged into it

Examples

```
route() |>
  route_add("HEAD", "*", function(...) {
    message("Someone's looking")
    TRUE
  }) |>
  route_merge(
    sizelimit_route()
  )
```

route_stack *Combine routes in a stack*

Description

This function allows you to combine multiple routes into a stack in order to dispatch on them until one of them returns `FALSE`. This allows you to have a router that can pass a request through multiple handlers before sending it along to the client or other middleware

Usage

```
route_stack(x, ...)

## Default S3 method:
route_stack(x, ...)

## S3 method for class 'Route'
route_stack(x, ...)

## S3 method for class 'AssetRoute'
route_stack(x, ...)

## S3 method for class 'RouteStack'
route_stack(x, ..., .after = NULL)
```

Arguments

x	A Route or RouteStack object
...	one or more named Route objects
.after	Where in the stack should the new routes be placed. NULL means place them at the end.

Value

A RouteStack object. If x is a RouteStack then this will be returned, modified.

Examples

```
# Create an empty route stack
route_stack()

# Stack a route with another, returning a RouteStack
route(all = list("*" = function(...) TRUE)) |>
  route_stack(
    limit = sizelimit_route()
  )
```

---

shared_secret_route *Reject requests not in possession of the correct shared secret*

Description

This route is a simple authentication method that limits requests based on whether they are in possession of an agreed upon shared secret. Be aware that if the request is send over HTTP then the secret will be visible to anyone intercepting the request. For this reason you should only use this route in combination with HTTPS or accept the probability that the secret is exposed. If no shared secret is provided with the request *or* if the shared secret doesn't match a `400L Bad Request` response is returned.

Usage

```
shared_secret_route(secret, header)
```

Arguments

secret	The secret to check for in a request
header	The name of the header to look for the secret

Value

A Route object

See Also

Other Route constructors: asset_route(), openapi_route(), resource_route(), sizelimit_route()

---

sizelimit_route *Limit the size of requests*

Description

This route is meant for being called prior to retrieving of the request body. It inspects the Content-Length header and determines if the request should be allowed to proceed. The limit can be made variable by supplying a function to the limit argument returning a numeric. If the Content-Length header is missing and the limit is not Inf the response will be set to 411 - Length Required, If the header exists but exceeds the limit the response will be set to 413 - Request Entity Too Large. Otherwise the route will return TRUE and leave the response unchanged.

Usage

```
sizelimit_route(limit = 5 * 1024^2, method = "all", path = "*")
```

Arguments

limit	Either a numeric or a function returning a numeric when called with the request
method	The method this route should respond to. Defaults to "all"
path	The URL path this route should respond to. Defaults to "*" (any path)

Value

A Route object

See Also

Other Route constructors: asset_route(), openapi_route(), resource_route(), shared_secret_route()

Examples

```
limit_route <- sizelimit_route() # Default 5Mb limit
rook <- fiery::fake_request('http://www.example.com', 'post',
                            headers = list(Content_Length = 30*1024^2))
req <- reqres::Request$new(rook)
limit_route$dispatch(req)
req$respond()
```

Index