Package 'routr'

October 14, 2022

Type Package

Title A Simple Router for HTTP and WebSocket Requests

Version 0.4.1

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, assertthat, uuid, reqres, stringi, tools, digest

RoxygenNote 7.2.1

Suggests testthat, covr, fiery

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

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

NeedsCompilation no

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

Repository CRAN

Date/Publication 2022-08-19 13:40:05 UTC

R topics documented:

`ressource_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

```
ressource_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 recieve 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.
`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 `digest::digest()` 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

Either `TRUE` if no file is found or `continue = TRUE` or `FALSE` if a file is found and `continue = FALSE`

See Also

Other Route constructors: `sizelimit_route()`

Examples

```
# Map package files
res_route <- ressource_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 *Create a route for dispatching on URL*

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().

Details

The path will strip the query string prior to assignment of the handler, can contain wildcards, and can be parameterised using the : prefix. If there are multiple matches of the request path the most specific will be chosen. Specificity is based on number of elements (most), number of parameters (least), and number of wildcards (least), in that order. Parameter values will be available in the keys argument passed to the handler, e.g. a path of /user/:user_id will provide list(user_id = 123) for a dispatch on /user/123 in the keys argument.

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. All calls to handlers will be wrapped in try() and if an exception is raised the response code will be set to 500 with the body of the response being the error message. Further processing of the request will be terminated. If a different error handling scheme is wanted it must be implemented within the handler (the standard approach is chosen to avoid handler errors resulting in a server crash).

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.

Initialization

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

Usage

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

Arguments

...	Handlers to add up front. Must be in the form of named lists where the names corresponds to paths and the elements

Methods

The following methods are accessible in a `Route` object:

`add_handler(method, path, 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.

`remove_handler(method, path)` Removes the handler assigned to the specified method and path. If no handler have been assigned it will throw a warning.

`get_handler(method, path)` Returns a handler already assigned to the specified method and path. If no handler have been assigned it will throw a warning.

`remap_handlers(.f)` 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.

`dispatch(request, ...)` 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.

Methods

Public methods:

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

Method `new()`:

Usage:

```
Route$new(...)
```

Method `print()`**:**

Usage:

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

Method `add_handler()`**:**

Usage:

```
Route$add_handler(method, path, handler)
```

Method `remove_handler()`**:**

Usage:

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

Method `get_handler()`**:**

Usage:

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

Method `remap_handlers()`**:**

Usage:

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

Method `dispatch()`**:**

Usage:

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

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 recieved')
```

```
      TRUE
    }
  )
)

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

RouteStack *Combine multiple routes for sequential routing*

Description

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

```
route <- 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_

Field

The following fields are accessible in a RouteStack object:

attach_to Either "request" (default), "header", or "message" that defines which event the router should be attached to when used as a fiery plugin.

name The plugin name (used by fiery). Will return '<attach_to>_routr' (e.g. 'request_routr' if attach_to == 'request')

Methods

The following methods are accessible in a RouteStack object:

add_route(route, name, after = NULL) 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.

has_route(name) Test if the routestack contains a route with the given name.

remove(name) Removes the route with the given name from the stack.

`dispatch(request, ...)` 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.

`on_error(fun)` Set the error handling function. This must be a function that accepts an `error`, `request`, and `reponse` argument. The error handler will be called if any of the route handlers throws an error and can be used to modify the `500` response before it is send back. By default, the error will be signaled using `message`

`on_attach(app, on_error = NULL, ...)` Method for use by `fiery` when attached as a plugin. Should not be called directly.

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 recieved 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.

Methods

Public methods:

- `RouteStack$new()`
- `RouteStack$print()`
- `RouteStack$add_route()`
- `RouteStack$get_route()`
- `RouteStack$has_route()`
- `RouteStack$remove_route()`
- `RouteStack$dispatch()`

- RouteStack$on_attach()
- RouteStack$on_error()
- RouteStack$clone()

Method `new()`**:**

Usage:

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

Method `print()`**:**

Usage:

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

Method `add_route()`**:**

Usage:

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

Method `get_route()`**:**

Usage:

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

Method `has_route()`**:**

Usage:

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

Method `remove_route()`**:**

Usage:

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

Method `dispatch()`**:**

Usage:

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

Method `on_attach()`**:**

Usage:

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

Method `on_error()`**:**

Usage:

```
RouteStack$on_error(fun)
```

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)
```

---

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)
```

Arguments

`limit`	Either a numeric or a function returning a numeric when called with the request

Value

TRUE if the request are allowed to proceed, or FALSE if it should be terminated

See Also

Other Route constructors: ressource_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