Package ‘ambiorix’

April 6, 2022

Title Web Framework Inspired by 'Express.js'

Version 2.1.0

Description A web framework inspired by 'express.js' to build any web service from multi-page websites to 'RESTful' application programming interfaces.

License GPL-3

Encoding UTF-8

RoxygenNote 7.1.2

Depends R (>= 4.1.0)

Imports fs, log, cli, glue, httpuv, methods, promises, jsonlite, websocket, assertthat

Suggests mime, readr, ggplot2, htmltools, commonmark, htmlwidgets, testthat (>= 3.0.0)

URL `https://github.com/devOpifex/ambiorix`, `https://ambiorix.dev`

BugReports `https://github.com/devOpifex/ambiorix/issues`

Config/testthat/edition 3

NeedsCompilation no

Author John Coene [aut, cre] (<`https://orcid.org/0000-0002-6637-4107`>), Opifex [fnd]

Maintainer John Coene <`jcoenep@gmail.com`>

Repository CRAN

Date/Publication 2022-04-06 18:42:29 UTC

R topics documented:

`Ambiorix` *Ambiorix*

Description

Web server.

Value

An object of class `Ambiorix` from which one can add routes, routers, and run the application.

Super class

`ambiorix::Routing` -> `Ambiorix`

Public fields

`not_found` 404 Response, must be a handler function that accepts the request and the response, by default uses `response_404()`.

`error` 500 response when the route errors, must a handler function that accepts the request and the response, by default uses `response_500()`.

`on_stop` Callback function to run when the app stops, takes no argument.

Active bindings

`port` Port to run the application.

`host` Host to run the application.

Methods

Public methods:

- `Ambiorix$new()`
- `Ambiorix$listen()`
- `Ambiorix$set_404()`
- `Ambiorix$static()`
- `Ambiorix$start()`
- `Ambiorix$serialiser()`
- `Ambiorix$stop()`
- `Ambiorix$print()`
- `Ambiorix$clone()`

Method `new()`:

Usage:

```
Ambiorix$new(
  host = getOption("ambiorix.host", "0.0.0.0"),
  port = getOption("ambiorix.port", NULL),
  log = getOption("ambiorix.logger", TRUE)
)
```

Arguments:

`host` A string defining the host.

`port` Integer defining the port, defaults to `ambiorix.port` option: uses a random port if `NULL`.

`log` Whether to generate a log of events.

Details: Define the webserver.

Method `listen()`:

Usage:

```
Ambiorix$listen(port)
```

Arguments:

`port` Port number.

Details: Specifies the port to listen on.

Examples:

```
app <- Ambiorix$new()

app$listen(3000L)

app$get("/", function(req, res){
 res$send("Using {ambiorix}!")
})

if(interactive())
 app$start()
```

Method `set_404()`**:**

Usage:

```
Ambiorix$set_404(handler)
```

Arguments:

`handler` Function that accepts the request and returns an object describing an httpuv response, e.g.: `response()`.

Details: Sets the 404 page.

Examples:

```
app <- Ambiorix$new()

app$set_404(function(req, res){
 res$send("Nothing found here")
})

app$get("/", function(req, res){
 res$send("Using {ambiorix}!")
})

if(interactive())
 app$start()
```

Method `static()`**:**

Usage:

```
Ambiorix$static(path, uri = "www")
```

Arguments:

`path` Local path to directory of assets.

`uri` URL path where the directory will be available.

Details: Static directories

Method `start()`**:**

Usage:

```
Ambiorix$start(port = NULL, host = NULL, open = interactive())
```

Arguments:

`port` Integer defining the port, defaults to `ambiorix.port` option: uses a random port if `NULL`.

`host` A string defining the host.

`open` Whether to open the app the browser.

Details: Start Start the webserver.

Examples:

```
app <- Ambiorix$new()

app$get("/", function(req, res){
 res$send("Using {ambiorix}!")
})

if(interactive())
 app$list(posrt = 3000L)
```

Method `serialiser()`**:**

Usage:

`Ambiorix$serialiser(handler)`

Arguments:

`handler` Function to use to serialise. This function should accept two arguments: the object to serialise and `...`.

Details: Define Serialiser

Examples:

```
app <- Ambiorix$new()

app$serialiser(function(data, ...){
 jsonlite::toJSON(x, ..., pretty = TRUE)
})

app$get("/", function(req, res){
 res$send("Using {ambiorix}!")
})

if(interactive())
 app$start()
```

Method `stop()`**:**

Usage:

`Ambiorix$stop()`

Details: Stop Stop the webserver.

Method `print()`**:**

Usage:

`Ambiorix$print()`

Details: Print

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

Usage:

`Ambiorix$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

Examples

```
app <- Ambiorix$new()

app$get("/", function(req, res){
 res$send("Using {ambiorix}!")
})

app$on_stop <- function(){
 cat("Bye!\n")
}

if(interactive())
 app$start()


## ------------------------------------------------
## Method `Ambiorix$listen`
## ------------------------------------------------

app <- Ambiorix$new()

app$listen(3000L)

app$get("/", function(req, res){
 res$send("Using {ambiorix}!")
})

if(interactive())
 app$start()

## ------------------------------------------------
## Method `Ambiorix$set_404`
## ------------------------------------------------

app <- Ambiorix$new()

app$set_404(function(req, res){
 res$send("Nothing found here")
})

app$get("/", function(req, res){
 res$send("Using {ambiorix}!")
})

if(interactive())
 app$start()

## ------------------------------------------------
## Method `Ambiorix$start`
## ------------------------------------------------

app <- Ambiorix$new()
```

```
app$get("/", function(req, res){
 res$send("Using {ambiorix}!")
})

if(interactive())
 app$list(posrt = 3000L)

## ------------------------------------------------
## Method `Ambiorix$serialiser`
## ------------------------------------------------

app <- Ambiorix$new()

app$serialiser(function(data, ...){
 jsonlite::toJSON(x, ..., pretty = TRUE)
})

app$get("/", function(req, res){
 res$send("Using {ambiorix}!")
})

if(interactive())
 app$start()
```

`as_cookie_parser` *Define a Cookie Parser*

Description

Identifies a function as a cookie parser (see example).

Usage

```
as_cookie_parser(fn)
```

Arguments

`fn`	A function that accepts a single argument, `req` the Request and returns the parsed cookie string, generally a `list`. Note that the original cookie string is available on the Request at the `HTTP_COOKIE` field, get it with: `req$HTTP_COOKIE`

Examples

```
func <- function(req) {
 req$HTTP_COOKIE
}

parser <- as_cookie_parser(func)
```

```
app <- Ambiorix$new()
app$use(parser)
```

`as_cookie_preprocessor` *Define a Cookie Preprocessor*

Description

Identifies a function as a cookie preprocessor.

Usage

```
as_cookie_preprocessor(fn)
```

Arguments

`fn`	A function that accepts the same arguments as the `cookie` method of the Response class (name, value, ...), and returns a modified `value`.

Examples

```
func <- \(name, value, ...) {
 sprintf("prefix.%s", value)
}

prep <- as_cookie_preprocessor(func)

app <- Ambiorix$new()
app$use(prep)
```

`as_path_to_pattern` *Path to pattern*

Description

identify a function as a path to pattern function; a function that accepts a path and returns a matching pattern.

Usage

```
as_path_to_pattern(path)
```

Arguments

`path`	A function that accepts a character vector of length 1 and returns another character vector of length 1.

as_renderer *Create a Renderer*

Description

Create a custom renderer.

Usage

```
as_renderer(fn)
```

Arguments

`fn`	A function that accepts two arguments, the full path to the `file` to render, and the `data` to render.

content *Content Headers*

Description

Convenient functions for more readable content type headers.

Usage

```
content_html()

content_plain()

content_json()

content_csv()

content_tsv()

content_protobuf()
```

Examples

```
list(
 "Content-Type",
 content_json()
)

if(FALSE)
 req$header(
  "Content-Type",
  content_json()
)
```

`create_dockerfile` *Dockerfile*

Description

Create the dockerfile required to run the application. The dockerfile created will install packages from RStudio Public Package Manager which comes with pre-built binaries that much improve the speed of building of Dockerfiles.

Usage

```
create_dockerfile(port, host = "0.0.0.0")
```

Arguments

`port, host`	Port and host to serve the application.

Details

Reads the `DESCRIPTION` file of the project to produce the `Dockerfile`.

Examples

```
## Not run: create_dockerfile()
```

`default_cookie_parser` *Cookie Parser*

Description

Parses the cookie string.

Usage

```
default_cookie_parser(req)
```

Arguments

`req`	A Request.

Value

A `list` of key value pairs or cookie values.

`forward` *Forward Method*

Description

Makes it such that the web server skips this method and uses the next one in line instead.

Usage

```
forward()
```

Value

An object of class `forward`.

Examples

```
app <- Ambiorix$new()

app$get("/next", function(req, res){
 forward()
})

app$get("/next", function(req, res){
 res$send("Hello")
})

if(interactive())
 app$start()
```

`import` *Import Files*

Description

Import all R-files in a directory.

Usage

```
import(...)
```

Arguments

`...`	Directory from which to import `.R` or `.r` files.

Value

Invisibly returns `NULL`.

Examples

```
## Not run: import("views")
```

`is_renderer_obj` *Is Renderer*

Description

Check whether an object is a renderer.

Usage

```
is_renderer_obj(obj)
```

Arguments

`obj`	Object to check.

Value

Boolean

`jobj` *JSON Object*

Description

Serialises an object to JSON in `res$render`.

Usage

```
jobj(obj)
```

Arguments

`obj`	Object to serialise.

`mockRequest` *Mock Request*

Description

Mock a request, used for tests.

Usage

```
mockRequest(cookie = "", query = "", path = "/")
```

Arguments

`cookie`	Cookie string.
`query`	Query string.
`path`	Path string.

Examples

```
mockRequest()
```

`new_log` *Logger*

Description

Returns a new logger using the `log` package.

Usage

```
new_log(prefix = ">", write = FALSE, file = "ambiorix.log", sep = "")
```

Arguments

`prefix`	String to prefix all log messages.
`write`	Whether to write the log to the `file`.
`file`	Name of the file to dump the logs to, only used if `write` is `TRUE`.
`sep`	Separator between `prefix` and other flags and messages.

Value

An R& of class `log::Logger`.

Examples

```
log <- new_log()
log$log("Hello world")
```

`parsers` *Parsers*

Description

Collection of parsers to translate request data.

Usage

```
parse_multipart(req)

parse_json(req, ...)
```

Arguments

`req`	The request object.
`...`	Additional arguments passed to the internal parsers.

Value

Returns the parsed value as a `list` or `NULL` if it failed to parse.

Functions

- `parse_multipart()`: Parse `multipart/form-data` using `mime::parse_multipart()`.
- `parse_json()`: Parse `multipart/form-data` using `jsonlite::fromJSON()`.

`pre_hook` *Pre Hook Response*

Description

Pre Hook Response

Usage

```
pre_hook(content, data)
```

Arguments

`content`	File content, a character vector.
`data`	A list of data passed to `glue::glue_data`.

Request *Request*

Description

A request.

Public fields

`HEADERS` Headers from the request.

`HTTP_ACCEPT` Content types to accept.

`HTTP_ACCEPT_ENCODING` Encoding of the request.

`HTTP_ACCEPT_LANGUAGE` Language of the request.

`HTTP_CACHE_CONTROL` Directorives for the cache (case-insensitive).

`HTTP_CONNECTION` Controls whether the network connection stays open after the current transaction finishes.

`HTTP_COOKIE` Cookie data.

`HTTP_HOST` Host making the request.

`HTTP_SEC_FETCH_DEST` Indicates the request's destination. That is the initiator of the original fetch request, which is where (and how) the fetched data will be used.

`HTTP_SEC_FETCH_MODE` Indicates mode of the request.

`HTTP_SEC_FETCH_SITE` Indicates the relationship between a request initiator's origin and the origin of the requested resource.

`HTTP_SEC_FETCH_USER` Only sent for requests initiated by user activation, and its value will always be `?1`.

`HTTP_UPGRADE_INSECURE_REQUESTS` Signals that server supports upgrade.

`HTTP_USER_AGENT` User agent.

`httpuv.version` Version of httpuv.

`PATH_INFO` Path of the request.

`QUERY_STRING` Query string of the request.

`REMOTE_ADDR` Remote address.

`REMOTE_PORT` Remote port.

`REQUEST_METHOD` Method of the request, e.g.: `GET`.

`rook.errors` Errors from rook.

`rook.input` Rook inputs.

`rook.url_scheme` Rook url scheme.

`rook.version` Rook version.

`SCRIPT_NAME` The initial portion of the request URL's "path" that corresponds to the application object, so that the application knows its virtual "location".

`SERVER_NAME` Server name.

`SERVER_PORT` Server port

`CONTENT_LENGTH` Size of the message body.

`CONTENT_TYPE` Type of content of the request.

`HTTP_REFERER` Contains an absolute or partial address of the page that makes the request.

`body` Request, an environment.

`query` Parsed `QUERY_STRING`, `list`.

`params` A `list` of parameters.

`cookie` Parsed `HTTP_COOKIE`.

Methods

Public methods:

- `Request$new()`
- `Request$print()`
- `Request$set()`
- `Request$get()`
- `Request$get_header()`
- `Request$parse_multipart()`
- `Request$parse_json()`
- `Request$clone()`

Method `new()`:

Usage:

```
Request$new(req)
```

Arguments:

`req` Original request (environment).

Details: Constructor

Method `print()`:

Usage:

```
Request$print()
```

Details: Print

Method `set()`:

Usage:

```
Request$set(name, value)
```

Arguments:

`name` Name of the variable.
`value` Value of the variable.

Details: Set Data

Returns: Invisible returns self.

Method `get()`**:**

Usage:

```
Request$get(name)
```

Arguments:

`name` Name of the variable to get.

Details: Get data

Method `get_header()`**:**

Usage:

```
Request$get_header(name)
```

Arguments:

`name` Name of the header

Details: Get Header

Method `parse_multipart()`**:**

Usage:

```
Request$parse_multipart()
```

Details: Parse Multipart encoded data

Method `parse_json()`**:**

Usage:

```
Request$parse_json(...)
```

Arguments:

`...` Arguments passed to `jsonlite::fromJSON()`.

Details: Parse JSON encoded data

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Response *Response*

Description

Response class to generate responses sent from the server.

Active bindings

`status` Status of the response, defaults to `200L`.

`headers` Named list of headers.

Methods

Public methods:

- `Response$set_status()`
- `Response$send()`
- `Response$sendf()`
- `Response$text()`
- `Response$send_file()`
- `Response$redirect()`
- `Response$render()`
- `Response$json()`
- `Response$csv()`
- `Response$tsv()`
- `Response$htmlwidget()`
- `Response$md()`
- `Response$png()`
- `Response$jpeg()`
- `Response$image()`
- `Response$ggplot2()`
- `Response$print()`
- `Response$set()`
- `Response$get()`
- `Response$header()`
- `Response$header_content_json()`
- `Response$header_content_html()`
- `Response$header_content_plain()`
- `Response$header_content_csv()`
- `Response$header_content_tsv()`
- `Response$get_headers()`
- `Response$get_header()`

- `Response$set_headers()`
- `Response$set_header()`
- `Response$pre_render_hook()`
- `Response$post_render_hook()`
- `Response$cookie()`
- `Response$clear_cookie()`
- `Response$clone()`

Method `set_status()`**:**

Usage:

```
Response$set_status(status)
```

Arguments:

`status` An integer defining the status.

Details: Set the status of the response.

Method `send()`**:**

Usage:

```
Response$send(body, headers = NULL, status = NULL)
```

Arguments:

`body` Body of the response.

`headers` HTTP headers to set.

`status` Status of the response, if `NULL` uses `self$status`.

Details: Send a plain HTML response.

Method `sendf()`**:**

Usage:

```
Response$sendf(body, ..., headers = NULL, status = NULL)
```

Arguments:

`body` Body of the response.

`...` Passed to `...` of `sprintf`.

`headers` HTTP headers to set.

`status` Status of the response, if `NULL` uses `self$status`.

Details: Send a plain HTML response, pre-processed with sprintf.

Method `text()`**:**

Usage:

```
Response$text(body, headers = NULL, status = NULL)
```

Arguments:

`body` Body of the response.

`headers` HTTP headers to set.

`status` Status of the response, if `NULL` uses `self$status`.

Details: Send a plain text response.

Method `send_file()`**:**

Usage:

```
Response$send_file(file, headers = NULL, status = NULL)
```

Arguments:

`file` File to send.

`headers` HTTP headers to set.

`status` Status of the response.

Details: Send a file.

Method `redirect()`**:**

Usage:

```
Response$redirect(path, status = NULL)
```

Arguments:

`path` Path or URL to redirect to.

`status` Status of the response, if `NULL` uses `self$status`.

Details: Redirect to a path or URL.

Method `render()`**:**

Usage:

```
Response$render(file, data = list(), headers = NULL, status = NULL)
```

Arguments:

`file` Template file.

`data` List to fill [% tags %].

`headers` HTTP headers to set.

`status` Status of the response, if `NULL` uses `self$status`.

Details: Render a template file.

Method `json()`**:**

Usage:

```
Response$json(body, headers = NULL, status = NULL, ...)
```

Arguments:

`body` Body of the response.

`headers` HTTP headers to set.

`status` Status of the response, if `NULL` uses `self$status`.

`...` Additional arguments passed to the serialiser.

Details: Render an object as JSON.

Method `csv()`**:**

Usage:

```
Response$csv(data, name = "data", status = NULL, ...)
```

Arguments:

`data` Data to convert to CSV.

`name` Name of the file.

`status` Status of the response, if `NULL` uses `self$status`.

`...` Additional arguments passed to `readr::format_csv()`.

Details: Sends a comma separated value file

Method `tsv()`**:**

Usage:

```
Response$tsv(data, name = "data", status = NULL, ...)
```

Arguments:

`data` Data to convert to CSV.

`name` Name of the file.

`status` Status of the response, if `NULL` uses `self$status`.

`...` Additional arguments passed to `readr::format_tsv()`.

Details: Sends a tab separated value file

Method `htmlwidget()`**:**

Usage:

```
Response$htmlwidget(widget, status = NULL, ...)
```

Arguments:

`widget` The widget to use.

`status` Status of the response, if `NULL` uses `self$status`.

`...` Additional arguments passed to `htmlwidgets::saveWidget()`.

Details: Sends an htmlwidget.

Method `md()`**:**

Usage:

```
Response$md(file, data = list(), headers = NULL, status = NULL)
```

Arguments:

`file` Template file.

`data` List to fill [% tags %].

`headers` HTTP headers to set.

`status` Status of the response, if `NULL` uses `self$status`.

Details: Render a markdown file.

Method `png()`**:**

Usage:

```
Response$png(file)
```

Arguments:

`file` Path to local file.

Details: Send a png file

Method `jpeg()`**:**

Usage:

```
Response$jpeg(file)
```

Arguments:

`file` Path to local file.

Details: Send a jpeg file

Method `image()`**:**

Usage:

```
Response$image(file)
```

Arguments:

`file` Path to local file.

Details: Send an image Similar to png and `jpeg` methods but guesses correct method based on file extension.

Method `ggplot2()`**:**

Usage:

```
Response$ggplot2(plot, ..., type = c("png", "jpeg"))
```

Arguments:

`plot` Ggplot2 plot object.

`...` Passed to `ggplot2::ggsave()`

`type` Type of image to save.

Details: Ggplot2

Method `print()`**:**

Usage:

```
Response$print()
```

Details: Print

Method `set()`**:**

Usage:

```
Response$set(name, value)
```

Arguments:

`name` Name of the variable.

`value` Value of the variable.

Details: Set Data

Returns: Invisible returns self.

Method `get()`**:**

Usage:

`Response$get(name)`

Arguments:

`name` Name of the variable to get.

Details: Get data

Method `header()`**:**

Usage:

```
Response$header(name, value)
```

Arguments:

`name, value` Name and value of the header.

Details: Add headers to the response.

Returns: Invisibly returns self.

Method `header_content_json()`**:**

Usage:

```
Response$header_content_json()
```

Details: Set Content Type to JSON

Returns: Invisibly returns self.

Method `header_content_html()`**:**

Usage:

```
Response$header_content_html()
```

Details: Set Content Type to HTML

Returns: Invisibly returns self.

Method `header_content_plain()`**:**

Usage:

```
Response$header_content_plain()
```

Details: Set Content Type to Plain Text

Returns: Invisibly returns self.

Method `header_content_csv()`**:**

Usage:

```
Response$header_content_csv()
```

Details: Set Content Type to CSV

Returns: Invisibly returns self.

Method `header_content_tsv()`**:**

Usage:

```
Response$header_content_tsv()
```

Details: Set Content Type to TSV

Returns: Invisibly returns self.

Method `get_headers()`**:**

Usage:

`Response$get_headers()`

Details: Get headers Returns the list of headers currently set.

Method `get_header()`**:**

Usage:

`Response$get_header(name)`

Arguments:

`name` Name of the header to return.

Details: Get a header Returns a single header currently, `NULL` if not set.

Method `set_headers()`**:**

Usage:

`Response$set_headers(headers)`

Arguments:

`headers` A named list of headers to set.

Details: Set headers

Method `set_header()`**:**

Usage:

`Response$set_header(name, value)`

Arguments:

`name` Name of the header.

`value` Value to set.

Details: Set a Header

Returns: Invisible returns self.

Method `pre_render_hook()`**:**

Usage:

`Response$pre_render_hook(hook)`

Arguments:

`hook` A function that accepts at least 4 arguments:

- `self`: The `Request` class instance.
- `content`: File content a vector of character string, content of the template.
- `data`: `list` passed from `render` method.
- `ext`: File extension of the template file.

This function is used to add pre-render hooks to the `render` method. The function should return an object of class `responsePreHook` as obtained by `pre_hook()`. This is meant to be used by middlewares to, if necessary, pre-process rendered data.

Include `...` in your `hook` to ensure it will handle potential updates to hooks in the future.

Details: Add a pre render hook. Runs before the `render` and `send_file` method.

Returns: Invisible returns self.

Method `post_render_hook()`**:**

Usage:

```
Response$post_render_hook(hook)
```

Arguments:

`hook` A function to run after the rendering of HTML. It should accept at least 3 arguments:
- `self`: The `Request` class instance.
- `content`: File content a vector of character string, content of the template.
- `ext`: File extension of the template file.

Include `...` in your `hook` to ensure it will handle potential updates to hooks in the future.

Details: Post render hook.

Returns: Invisible returns self.

Method `cookie()`**:**

Usage:

```
Response$cookie(
  name,
  value,
  expires = getOption("ambiorix.cookie.expire"),
  max_age = getOption("ambiorix.cookie.maxage"),
  domain = getOption("ambiorix.cookie.domain"),
  path = getOption("ambiorix.cookie.path", "/"),
  secure = getOption("ambiorix.cookie.secure", TRUE),
  http_only = getOption("ambiorix.cookie.httponly", TRUE),
  same_site = getOption("ambiorix.cookie.savesite")
)
```

Arguments:

`name` Name of the cookie.

`value` value of the cookie.

`expires` Expiry, if an integer assumes it's the number of seconds from now. Otherwise accepts an object of class `POSIXct` or `Date`. If a `character` string then it is set as-is and not pre-processed. If unspecified, the cookie becomes a session cookie. A session finishes when the client shuts down, after which the session cookie is removed.

`max_age` Indicates the number of seconds until the cookie expires. A zero or negative number will expire the cookie immediately. If both `expires` and `max_age` are set, the latter has precedence.

`domain` Defines the host to which the cookie will be sent. If omitted, this attribute defaults to the host of the current document URL, not including subdomains.

`path` Indicates the path that must exist in the requested URL for the browser to send the Cookie header.

`secure` Indicates that the cookie is sent to the server only when a request is made with the https: scheme (except on localhost), and therefore, is more resistant to man-in-the-middle attacks.

`http_only` Forbids JavaScript from accessing the cookie, for example, through the document.cookie property.

`same_site` Controls whether or not a cookie is sent with cross-origin requests, providing some protection against cross-site request forgery attacks (CSRF). Accepts `Strict`, `Lax`, or `None`.

Details: Set a cookie Overwrites existing cookie of the same `name`.

Returns: Invisibly returns self.

Method `clear_cookie()`**:**

Usage:

```
Response$clear_cookie(name)
```

Arguments:

`name` Name of the cookie to clear.

Details: Clear a cookie Clears the value of a cookie.

Returns: Invisibly returns self.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`responses` *Plain Responses*

Description

Plain HTTP Responses.

Usage

```
response(body, headers = list(), status = 200L)

response_404(
  body = "404: Not found",
  headers = list(`Content-Type` = content_html()),
  status = 404L
)

response_500(
  body = "500: Server Error",
  headers = list(`Content-Type` = content_html()),
  status = 500L
)
```

Arguments

`body`	Body of response.
`headers`	HTTP headers.
`status`	Response status

Examples

```
app <- Ambiorix$new()

# html
app$get("/", function(req, res){
 res$send("hello!")
})

# text
app$get("/text", function(req, res){
 res$text("hello!")
})

if(interactive())
 app$start()
```

`robj` *R Object*

Description

Treats a data element rendered in a response (`res$render`) as a data object and ultimately uses `dput()`.

Usage

```
robj(obj)
```

Arguments

`obj`	R object to treat.

Details

For instance in a template, x <- [% var %] will not work with `res$render(data=list(var = "hello"))` because this will be replace like `x <-hello` (missing quote): breaking the template. Using `robj` one would obtain `x <-"hello"`.

`Router` *Router*

Description

Web server.

Super class

`ambiorix::Routing` -> `Router`

Public fields

`error` 500 response when the route errors, must a handler function that accepts the request and the response, by default uses `response_500()`.

Methods

Public methods:

- `Router$new()`
- `Router$print()`
- `Router$clone()`

Method `new()`**:**

Usage:

```
Router$new(path)
```

Arguments:

`path` The base path of the router.

Details: Define the base route.

Method `print()`**:**

Usage:

```
Router$print()
```

Details: Print

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# log
logger <- new_log()
# router
# create router
router <- Router$new("/users")

router$get("/", function(req, res){
 res$send("List of users")
})

router$get("/:id", function(req, res){
 logger$log("Return user id:", req$params$id)
 res$send(req$params$id)
})

router$get("/:id/profile", function(req, res){
 msg <- sprintf("This is the profile of user #%s", req$params$id)
 res$send(msg)
})

# core app
app <- Ambiorix$new()

app$get("/", function(req, res){
 res$send("Home!")
})

# mount the router
app$use(router)

if(interactive())
 app$start()
```

`Routing` *Core Routing Class*

Description

Core routing class. Do not use directly, see Ambiorix, and Router.

Public fields

`error` Error handler.

Methods

Public methods:

- `Routing$new()`
- `Routing$get()`
- `Routing$put()`
- `Routing$patch()`
- `Routing$delete()`
- `Routing$post()`
- `Routing$options()`
- `Routing$all()`
- `Routing$receive()`
- `Routing$print()`
- `Routing$use()`
- `Routing$get_routes()`
- `Routing$get_receivers()`
- `Routing$get_middleware()`
- `Routing$clone()`

Method `new()`**:**

Usage:

```
Routing$new(path = "")
```

Arguments:

`path` Prefix path.

Details: Initialise

Method `get()`**:**

Usage:

```
Routing$get(path, handler, error = NULL)
```

Arguments:

`path` Route to listen to, `:` defines a parameter.

`handler` Function that accepts the request and returns an object describing an httpuv response, e.g.: `response()`.

`error` Handler function to run on error.

Details: GET Method
Add routes to listen to.

Examples:

```
app <- Ambiorix$new()

app$get("/", function(req, res){
 res$send("Using {ambiorix}!")
})

if(interactive())
 app$start()
```

Method `put()`**:**

Usage:

```
Routing$put(path, handler, error = NULL)
```

Arguments:

`path` Route to listen to, `:` defines a parameter.

`handler` Function that accepts the request and returns an object describing an httpuv response, e.g.: `response()`.

`error` Handler function to run on error.

Details: PUT Method
Add routes to listen to.

Method `patch()`**:**

Usage:

```
Routing$patch(path, handler, error = NULL)
```

Arguments:

`path` Route to listen to, `:` defines a parameter.

`handler` Function that accepts the request and returns an object describing an httpuv response, e.g.: `response()`.

`error` Handler function to run on error.

Details: PATCH Method
Add routes to listen to.

Method `delete()`**:**

Usage:

```
Routing$delete(path, handler, error = NULL)
```

Arguments:

`path` Route to listen to, `:` defines a parameter.

`handler` Function that accepts the request and returns an object describing an httpuv response, e.g.: `response()`.

`error` Handler function to run on error.

Details: DELETE Method
Add routes to listen to.

Method `post()`**:**

Usage:

```
Routing$post(path, handler, error = NULL)
```

Arguments:

`path` Route to listen to.

`handler` Function that accepts the request and returns an object describing an httpuv response, e.g.: `response()`.

`error` Handler function to run on error.

Details: POST Method
Add routes to listen to.

Method `options()`**:**

Usage:

```
Routing$options(path, handler, error = NULL)
```

Arguments:

`path` Route to listen to.

`handler` Function that accepts the request and returns an object describing an httpuv response, e.g.: `response()`.

`error` Handler function to run on error.

Details: OPTIONS Method
Add routes to listen to.

Method `all()`**:**

Usage:

```
Routing$all(path, handler, error = NULL)
```

Arguments:

`path` Route to listen to.

`handler` Function that accepts the request and returns an object describing an httpuv response, e.g.: `response()`.

`error` Handler function to run on error.

Details: All Methods
Add routes to listen to for all methods `GET`, `POST`, `PUT`, `DELETE`, and `PATCH`.

Method `receive()`**:**

Usage:

```
Routing$receive(name, handler)
```

Arguments:

`name` Name of message.

`handler` Function to run when message is received.

Details: Receive Websocket Message

Examples:

```
app <- Ambiorix$new()

app$get("/", function(req, res){
 res$send("Using {ambiorix}!")
})

app$receive("hello", function(msg, ws){
 print(msg) # print msg received

 # send a message back
 ws$send("hello", "Hello back! (sent from R)")
})

if(interactive())
 app$start()
```

Method `print()`**:**

Usage:

`Routing$print()`

Details: Print

Method `use()`**:**

Usage:

`Routing$use(use)`

Arguments:

`use` Either a router as returned by Router, a function to use as middleware, or a `list` of functions. If a function is passed, it must accept two arguments (the request, and the response): this function will be executed every time the server receives a request. *Middleware may but does not have to return a response, unlike other methods such as* `get` Note that multiple routers and middlewares can be used.

Details: Use a router or middleware

Method `get_routes()`**:**

Usage:

`Routing$get_routes()`

Details: Get the routes

Method `get_receivers()`**:**

Usage:

`Routing$get_receivers()`

Details: Get the receivers

Method `get_middleware()`**:**

Usage:

`Routing$get_middleware()`

Details: Get the middleware

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

Usage:

`Routing$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

Examples

```
## ------------------------------------------------
## Method `Routing$get`
## ------------------------------------------------

app <- Ambiorix$new()

app$get("/", function(req, res){
 res$send("Using {ambiorix}!")
})

if(interactive())
 app$start()

## ------------------------------------------------
## Method `Routing$receive`
## ------------------------------------------------

app <- Ambiorix$new()

app$get("/", function(req, res){
 res$send("Using {ambiorix}!")
})

app$receive("hello", function(msg, ws){
 print(msg) # print msg received

 # send a message back
 ws$send("hello", "Hello back! (sent from R)")
})

if(interactive())
 app$start()
```

`serialise` *Serialise to JSON*

Description

Serialise an object to JSON. Default serialiser can be change by setting the `AMBIORIX_SERIALISER` option to the desired function.

Usage

```
serialise(data, ...)
```

Arguments

`data`	Data to serialise.
`...`	Passed to serialiser.

Examples

```
## Not run: serialise(cars)
```

`set_log` *Customise logs*

Description

Customise the internal logs used by Ambiorix.

Usage

```
set_log_info(log)

set_log_success(log)

set_log_error(log)
```

Arguments

`log`	An object of class `Logger`, see log::Logger.

`set_params` *Set Parameters*

Description

Set the query's parameters.

Usage

```
set_params(path, route = NULL)
```

Arguments

`path`	Correspond's the the requests' `PATH_INFO`
`route`	See `Route`

Value

Parameter list

`stop_all` *Stop*

Description

Stop all servers.

Usage

```
stop_all()
```

`token_create` *Token*

Description

Create a token

Usage

```
token_create(n = 16L)
```

Arguments

`n`	Number of bytes.

`Websocket` *Websocket*

Description

Handle websocket messages.

Methods

Public methods:

- `Websocket$new()`
- `Websocket$send()`
- `Websocket$print()`
- `Websocket$clone()`

Method `new()`**:**

Usage:

`Websocket$new(ws)`

Arguments:

`ws`

Details: Constructor

Method `send()`**:**

Usage:

`Websocket$send(name, message)`

Arguments:

`name` Name, identifier, of the message.

`message` Content of the message, anything that can be serialised to JSON.

Details: Send a message

Method `print()`**:**

Usage:

`Websocket$print()`

Details: Print

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

Usage:

`Websocket$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

`websocket_client` *Websocket Client*

Description

Handle ambiorix websocket client.

Usage

```
copy_websocket_client(path)

get_websocket_client()
```

Arguments

`path`	Path to copy the file to.

Functions

- `copy_websocket_client` Copies the websocket client file, useful when ambiorix was not setup with the ambiorix generator.
- `get_websocket_client` Retrieves the full path to the local websocket client.

Index