

Package: ambiorix (via r-universe)

October 17, 2026

Title Web Framework Inspired by 'Express.js'

Version 3.1.0

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

LazyData false

Roxygen list(markdown = TRUE)

Imports log, cli, glue, httpuv, methods, assertthat, webutils (>= 1.2.0), yyjsonr (>= 0.1.20), mime (>= 0.13), promises (>= 1.5.0)

Suggests readr, readxl, ggplot2, jsonlite, websocket, htmltools, commonmark, htmlwidgets, testthat (>= 3.0.0)

URL <https://github.com/ambiorix-web/ambiorix>, <https://ambiorix.dev>

BugReports <https://github.com/ambiorix-web/ambiorix/issues>

Config/testthat/edition 3

Config/roxygen2/version 8.0.0

Config/pak/sysreqs make libssl-dev zlib1g-dev

Repository <https://ambiorix-web.r-universe.dev>

Date/Publication 2026-07-19 17:34:12 UTC

RemoteUrl <https://github.com/ambiorix-web/ambiorix>

RemoteRef HEAD

RemoteSha 36a90a122fe63899b7934f4cc4d231b4a719d512

Contents

Ambiorix	2
as_cookie_parser	10

as_cookie_preprocessor	11
as_path_to_pattern	12
as_renderer	13
content	13
default_cookie_parser	14
forward	15
import	16
jobj	16
mockRequest	17
new_log	17
openapi-schemas	18
openapi_docs	19
openapi_param	20
openapi_parameters	21
openapi_request_body	22
openapi_response	22
openapi_responses	23
parse_form_urlencoded	24
parse_json	28
parse_multipart	32
pre_hook	36
Request	37
Response	40
responses	48
robject	49
Router	50
Routing	51
routing-http-methods	56
serialise	58
set_log	59
stop_all	60
token_create	61
use_html_template	61
Websocket	62
websocket_client	63

Index	65
--------------	-----------

Ambiorix

Ambiorix

Description

Web server.

Value

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

Super class

`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.

`limit` Max body size, defaults to $5 * 1024 * 1024$.

Methods

Public methods:

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

`Ambiorix$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.

Ambiorix\$cache_templates():

Usage:

Ambiorix\$cache_templates()

Details: Cache templates in memory instead of reading them from disk.

Ambiorix\$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()
```

Ambiorix\$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()
```

Ambiorix\$set_error():

Usage:

Ambiorix\$set_error(handler)

Arguments:

handler Function that accepts a request, response and an error object.

Details: Sets the error handler.

Examples:

```
# my custom error handler:
error_handler <- function(req, res, error) {
  if (!is.null(error)) {
    error_msg <- conditionMessage(error)
    cli::cli_alert_danger("Error: {error_msg}")
  }
  response <- list(
    code = 500L,
    msg = "Uhhmmm... Looks like there's an error from our side :("
  )
  res$
    set_status(500L)$
    json(response)
}

# handler for GET at /whoami:
whoami <- function(req, res) {
  # simulate error (object 'Pikachu' is not defined)
  print(Pikachu)
}

app <- Ambiorix$
  new()$
  set_error(error_handler)$
  get("/whoami", whoami)

if (interactive()) {
  app$start(open = FALSE)
}
```

Ambiorix\$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

Ambiorix\$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 the webserver.

Examples:

```
app <- Ambiorix$new()

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

if(interactive())
  app$start(port = 3000L)
```

`Ambiorix$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()
```

`Ambiorix$openapi():`

Usage:

```
Ambiorix$openapi(
  title = "API",
  version = "1.0.0",
  description = NULL,
```

```

    ui_path = "/docs",
    spec_path = "/openapi.json",
    assets_path = "/__swagger__",
    ...
)

```

Arguments:

`title` Title of the API.

`version` Version of the API.

`description` Optional description of the API.

`ui_path` Path at which the Swagger UI is served.

`spec_path` Path at which the OpenAPI JSON document is served.

`assets_path` Path at which the Swagger UI assets (CSS & JavaScript) are served.

... Additional fields added to the OpenAPI info object.

Details: Enable OpenAPI (Swagger) documentation.

When enabled, two routes are registered when the app starts: an interactive Swagger UI (`ui_path`, default `/docs`) and the OpenAPI JSON document (`spec_path`, default `/openapi.json`). Only routes registered with a `docs` argument (see [openapi_docs\(\)](#)) appear in the document.

The Swagger UI assets (CSS & JavaScript) are bundled with ambiorix and served locally at `assets_path` (default `/__swagger__`), so the docs work without an internet connection.

If `ui_path` or `spec_path` collides with an existing route, or `assets_path` collides with an existing static directory, the corresponding docs route (or asset directory) is not registered and a warning is emitted. The OpenAPI document is always serialised with the default serialiser, regardless of any custom serialiser set via `serialiser()`.

Examples:

```

app <- Ambiorix$new()

app$openapi(title = "My API", version = "1.0.0")

app$get(
  "/",
  function(req, res) {
    res$send("Using {ambiorix}!")
  },
  docs = openapi_docs(
    summary = "Landing page",
    responses = openapi_responses(
      openapi_response(200, "The landing page")
    )
  )
)

if (interactive())
  app$start()

```

`Ambiorix$stop():`

Usage:

```
Ambiorix$stop()
```

Details: Stop Stop the webserver.

```
Ambiorix$print():
```

Usage:

```
Ambiorix$print()
```

Details: Print

`Ambiorix$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$set_error()`
## -----

# my custom error handler:
error_handler <- function(req, res, error) {
  if (!is.null(error)) {
    error_msg <- conditionMessage(error)
    cli::cli_alert_danger("Error: {error_msg}")
  }
  response <- list(
    code = 500L,
    msg = "Uhhmmm... Looks like there's an error from our side :("
  )
  res$
    set_status(500L)$
    json(response)
}

# handler for GET at /whoami:
whoami <- function(req, res) {
  # simulate error (object 'Pikachu' is not defined)
  print(Pikachu)
}

app <- Ambiorix$
  new()$
  set_error(error_handler)$
  get("/whoami", whoami)

if (interactive()) {
  app$start(open = FALSE)
}

## -----
## Method `Ambiorix$start()`
## -----

app <- Ambiorix$new()

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

```

```

    res$send("Using {ambiorix}!")
  })

  if(interactive())
    app$start(port = 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()

  ## -----
  ## Method `Ambiorix$openapi()`
  ## -----

  app <- Ambiorix$new()

  app$openapi(title = "My API", version = "1.0.0")

  app$get(
    "/",
    function(req, res) {
      res$send("Using {ambiorix}!")
    },
    docs = openapi_docs(
      summary = "Landing page",
      responses = openapi_responses(
        openapi_response(200, "The landing page")
      )
    )
  )

  if (interactive())
    app$start()

```

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

Value

Object of class "cookieParser".

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.

Value

Object of class "cookiePreprocessor".

Examples

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

prep <- as_cookie_preprocessor(func)

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

as_path_to_pattern	<i>Path to pattern</i>
--------------------	------------------------

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.
------	--

Value

Object of class "pathToPattern".

Examples

```
fn <- function(path) {
  pattern <- gsub(":[^/]+", "(\\\\w+)", path)
  paste0("^", pattern, "$")
}

path_to_pattern <- as_path_to_pattern(fn)

path <- "/dashboard/profile/:user_id"
pattern <- path_to_pattern(path) # "^/dashboard/profile/(\\w+)$"
```

as_renderer	<i>Create a Renderer</i>
-------------	--------------------------

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.
----	---

Value

A renderer function.

Examples

```
if (interactive()) {  
  fn <- function(path, data) {  
    # ...  
  }  
  
  as_renderer(fn)  
}
```

content	<i>Content Headers</i>
---------	------------------------

Description

Convenient functions for more readable content type headers.

Usage

```
content_html()  
  
content_plain()  
  
content_json()  
  
content_csv()
```

```
content_tsv()
```

```
content_protobuf()
```

Value

Length 1 character vector.

Examples

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

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

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.

Examples

```
if (interactive()) {  
  library(ambiorix)  
  
  #' Handle GET at '/greet'  
  #'  
  #' @export  
  say_hello <- function(req, res) {
```

```
    cookies <- default_cookie_parser(req)
    print(cookies)

    res$send("hello there!")
  }

  app <- Ambiorix$new()
  app$get("/greet", say_hello)
  app$start()
}
```

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

```
if (interactive()) {  
  import("views")  
}
```

`jobj`*JSON Object*

Description

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

Usage

```
jobj(obj)
```

Arguments

obj Object to serialise.

Value

Object of class "jobj".

Examples

```
if (interactive()) {  
  l <- list(a = "hello", b = 2L, c = 3)  
  jobj(l)  
}
```

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.

Value

A Request object.

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

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

Value

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

Examples

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

openapi-schemas	<i>OpenAPI Schemas</i>
-----------------	------------------------

Description

Build **OpenAPI Schema Objects** used to describe request bodies, responses, and parameters.

Usage

```
openapi_schema_string()
openapi_schema_integer()
openapi_schema_number()
openapi_schema_boolean()
openapi_schema_array(items)
openapi_schema_object(...)
```

Arguments

<code>items</code>	An OpenAPI schema (from any <code>openapi_schema_*</code> () helper) describing the type of every element in the array.
<code>...</code>	Named OpenAPI schemas describing the properties of an object.

Value

An object of class `ambiorix_openapi_schema`; a list that mirrors an OpenAPI schema object.

Examples

```
openapi_schema_string()

openapi_schema_object(
  id = openapi_schema_integer(),
  name = openapi_schema_string(),
  tags = openapi_schema_array(openapi_schema_string())
)
```

openapi_docs

OpenAPI Route Documentation

Description

Document a single route. Pass the result to the docs argument of a routing method (see [routing-http-methods](#)).

Usage

```
openapi_docs(
  summary = NULL,
  description = NULL,
  tags = NULL,
  parameters = NULL,
  request_body = NULL,
  responses = NULL
)
```

Arguments

summary	Short summary of what the route does.
description	Longer description of the route.
tags	Character vector of tags used to group routes.
parameters	Query, path, header, and cookie parameters; see openapi_parameters() .
request_body	The request body; see openapi_request_body() .
responses	The responses; see openapi_responses() .

Details

Path parameters are documented automatically from the route's `:param` tokens with a string schema, so only query, header, and cookie parameters need to be declared via [openapi_parameters\(\)](#). To override an automatic path parameter (e.g. to document it as an integer), declare it with [openapi_param\(\)](#) using `location = "path"` and a name matching the route token.

Value

An object of class `ambiorix_openapi_docs`.

See Also

[routing-http-methods](#)

Examples

```
openapi_docs(
    summary = "Get a user by ID",
    tags = "users",
    responses = openapi_responses(
        openapi_response(200, "The user")
    )
)
```

openapi_param

OpenAPI Parameter

Description

Describe a single query, path, header, or cookie parameter. Path parameters are documented automatically from the route's `:param` tokens with a string schema; declare one here with `location = "path"` to override that default, e.g. to document it as an integer.

Usage

```
openapi_param(
    name,
    location = c("query", "path", "header", "cookie"),
    required = FALSE,
    description = NULL,
    schema = openapi_schema_string()
)
```

Arguments

<code>name</code>	Name of the parameter. For path parameters this must match one of the route's <code>:param</code> tokens.
<code>location</code>	Where the parameter is passed; one of "query", "path", "header", or "cookie".
<code>required</code>	Whether the parameter is required. Path parameters are always required: this argument is ignored for them and forced to TRUE.
<code>description</code>	Human readable description of the parameter.
<code>schema</code>	An OpenAPI schema (see openapi-schemas) describing the parameter's type.

Value

An object of class `ambiorix_openapi_parameter`.

Examples

```
openapi_param(  
  "verbose",  
  location = "query",  
  description = "Return extra fields",  
  schema = openapi_schema_boolean()  
)  
  
# override the automatic string schema of a path parameter  
openapi_param(  
  "id",  
  location = "path",  
  schema = openapi_schema_integer()  
)
```

openapi_parameters	<i>OpenAPI Parameters</i>
--------------------	---------------------------

Description

Collect query, path, header, and cookie parameters for a route. Path parameters are documented automatically from the route's `:param` tokens with a string schema; declare one here to override that default.

Usage

```
openapi_parameters(...)
```

Arguments

... Objects created with `openapi_param()`.

Value

An object of class `ambiorix_openapi_parameters`.

Examples

```
openapi_parameters(  
  openapi_param("verbose", location = "query", schema = openapi_schema_boolean()),  
  openapi_param("X-Trace", location = "header")  
)
```

openapi_request_body *OpenAPI Request Body*

Description

Describe the body accepted by a route.

Usage

```
openapi_request_body(  
    schema,  
    content_type = "application/json",  
    required = TRUE,  
    description = NULL  
)
```

Arguments

schema	An OpenAPI schema (see openapi-schemas) describing the body.
content_type	The media type of the body.
required	Whether the body is required.
description	Human readable description of the body.

Value

An object of class `ambiorix_openapi_request_body`.

Examples

```
openapi_request_body(  
  schema = openapi_schema_object(  
    name = openapi_schema_string()  
  )  
)
```

openapi_response *OpenAPI Response*

Description

Describe a single response for a route.

Usage

```
openapi_response(  
  status,  
  description,  
  content_type = "application/json",  
  schema = NULL  
)
```

Arguments

- status HTTP status code, e.g. 200L. Also accepts the string "default" or a status range such as "2XX", as allowed by the OpenAPI specification.
- description Human readable description of the response.
- content_type The media type of the response body.
- schema An optional OpenAPI schema (see [openapi-schemas](#)) describing the response body.

Value

An object of class ambiorix_openapi_response.

Examples

```
openapi_response(  
  200,  
  "The user",  
  schema = openapi_schema_object(id = openapi_schema_integer())  
)
```

openapi_responses	<i>OpenAPI Responses</i>
-------------------	--------------------------

Description

Collect the responses of a route.

Usage

```
openapi_responses(...)
```

Arguments

- ... Objects created with [openapi_response\(\)](#).

Value

An object of class ambiorix_openapi_responses.

Examples

```
openapi_responses(  
  openapi_response(200, "Success"),  
  openapi_response(404, "Not found")  
)
```

parse_form_urlencoded *Parse application/x-www-form-urlencoded data*

Description

This function parses application/x-www-form-urlencoded data, typically used in form submissions.

Usage

```
parse_form_urlencoded(req, ...)
```

Arguments

req	The request object.
...	Additional parameters passed to the parser function.

Details

Overriding Default Parser:

By default, `parse_form_urlencoded()` uses `webutils::parse_http()`. You can override this globally by setting the `AMBIORIX_FORM_URLENCODED_PARSER` option:

```
options(AMBIORIX_FORM_URLENCODED_PARSER = my_other_custom_parser)
```

Your custom parser function *MUST* accept the following parameters:

1. body: Raw vector containing the form data.
2. ...: Additional optional parameters.

Value

A list of parsed form fields, with each key representing a form field name and each value representing the form field's value.

Named list

See Also

[parse_multipart\(\)](#), [parse_json\(\)](#)

Examples

```

if (interactive()) {
  library(ambiorix)
  library(htmltools)
  library(readxl)

  page_links <- function() {
    Map(
      f = function(href, label) {
        tags$a(href = href, label)
      },
      c("/", "/about", "/contact"),
      c("Home", "About", "Contact")
    )
  }

  forms <- function() {
    form1 <- tags$form(
      action = "/url-form-encoded",
      method = "POST",
      enctype = "application/x-www-form-urlencoded",
      tags$h4("form-url-encoded:"),
      tags$label(`for` = "first_name", "First Name"),
      tags$input(id = "first_name", name = "first_name", value = "John"),
      tags$label(`for` = "last_name", "Last Name"),
      tags$input(id = "last_name", name = "last_name", value = "Coene"),
      tags$button(type = "submit", "Submit")
    )

    form2 <- tags$form(
      action = "/multipart-form-data",
      method = "POST",
      enctype = "multipart/form-data",
      tags$h4("multipart/form-data:"),
      tags$label(`for` = "email", "Email"),
      tags$input(id = "email", name = "email", value = "john@mail.com"),
      tags$label(`for` = "framework", "Framework"),
      tags$input(id = "framework", name = "framework", value = "ambiorix"),
      tags$label(`for` = "file", "Upload CSV file"),
      tags$input(type = "file", id = "file", name = "file", accept = ".csv"),
      tags$label(`for` = "file2", "Upload xlsx file"),
      tags$input(type = "file", id = "file2", name = "file2", accept = ".xlsx"),
      tags$button(type = "submit", "Submit")
    )

    tagList(form1, form2)
  }

  home_get <- function(req, res) {
    html <- tagList(
      page_links(),
      tags$h3("hello, world!"),

```

```

    forms()
  )

  res$send(html)
}

home_post <- function(req, res) {
  body <- parse_json(req)
  # print(body)

  response <- list(
    code = 200L,
    msg = "hello, world"
  )
  res$json(response)
}

url_form_encoded_post <- function(req, res) {
  body <- parse_form_urlencoded(req)
  # print(body)

  list_items <- lapply(
    X = names(body),
    FUN = function(nm) {
      tags$li(
        nm,
        ":",
        body[[nm]]
      )
    }
  )
  input_vals <- tags$ul(list_items)

  html <- tagList(
    page_links(),
    tags$h3("Request processed"),
    input_vals
  )

  res$send(html)
}

multipart_form_data_post <- function(req, res) {
  body <- parse_multipart(req)

  list_items <- lapply(
    X = names(body),
    FUN = function(nm) {
      field <- body[[nm]]

      # if 'field' is a file, parse it & print on console:
      is_file <- "filename" %in% names(field)
      is_csv <- is_file && identical(field[["content_type"]], "text/csv")
    }
  )

```

```

    is_xlsx <- is_file &&
      identical(
        field[["content_type"]],
        "application/vnd.openxmlformats-officedocument.spreadsheetml.sheet"
      )

    if (is_file) {
      file_path <- tempfile()
      writeBin(object = field$value, con = file_path)
      on.exit(unlink(x = file_path))
    }

    if (is_csv) {
      # print(read.csv(file = file_path))
    }

    if (is_xlsx) {
      # print(readxl::read_xlsx(path = file_path))
    }

    tags$li(
      nm,
      ":",
      if (is_file) "printed on console" else field
    )
  }
)
input_vals <- tags$ul(list_items)

html <- tagList(
  page_links(),
  tags$h3("Request processed"),
  input_vals
)

res$send(html)
}

about_get <- function(req, res) {
  html <- tagList(
    page_links(),
    tags$h3("About Us")
  )
  res$send(html)
}

contact_get <- function(req, res) {
  html <- tagList(
    page_links(),
    tags$h3("Get In Touch!")
  )
  res$send(html)
}

```

```

app <- Ambiorix$new(port = 5000L)

app$
  get("/", home_get)$
  post("/", home_post)$
  get("/about", about_get)$
  get("/contact", contact_get)$
  post("/url-form-encoded", url_form_encoded_post)$
  post("/multipart-form-data", multipart_form_data_post)

app$start()
}

```

parse_json

Parse application/json data

Description

This function parses JSON data from the request body.

Usage

```
parse_json(req, ...)
```

Arguments

req	The request object.
...	Additional parameters passed to the parser function.

Details

Overriding Default Parser:

By default, `parse_json()` uses `yyjsonr::read_json_raw()` for JSON parsing. You can override this globally by setting the `AMBIORIX_JSON_PARSER` option:

```

my_json_parser <- function(body, ...) {
  txt <- rawToChar(body)
  jsonlite::fromJSON(txt, ...)
}
options(AMBIORIX_JSON_PARSER = my_json_parser)

```

Your custom parser *MUST* accept the following parameters:

1. `body`: Raw vector containing the JSON data.
2. `...`: Additional optional parameters.

Value

An R object (e.g., list or data frame) parsed from the JSON data.

Named list

See Also

[parse_multipart\(\)](#), [parse_form_urlencoded\(\)](#)

Examples

```
if (interactive()) {
  library(ambiorix)
  library(htmltools)
  library(readxl)

  page_links <- function() {
    Map(
      f = function(href, label) {
        tags$a(href = href, label)
      },
      c("/", "/about", "/contact"),
      c("Home", "About", "Contact")
    )
  }

  forms <- function() {
    form1 <- tags$form(
      action = "/url-form-encoded",
      method = "POST",
      enctype = "application/x-www-form-urlencoded",
      tags$h4("form-url-encoded:"),
      tags$label(`for` = "first_name", "First Name"),
      tags$input(id = "first_name", name = "first_name", value = "John"),
      tags$label(`for` = "last_name", "Last Name"),
      tags$input(id = "last_name", name = "last_name", value = "Coene"),
      tags$button(type = "submit", "Submit")
    )

    form2 <- tags$form(
      action = "/multipart-form-data",
      method = "POST",
      enctype = "multipart/form-data",
      tags$h4("multipart/form-data:"),
      tags$label(`for` = "email", "Email"),
      tags$input(id = "email", name = "email", value = "john@mail.com"),
      tags$label(`for` = "framework", "Framework"),
      tags$input(id = "framework", name = "framework", value = "ambiorix"),
      tags$label(`for` = "file", "Upload CSV file"),
      tags$input(type = "file", id = "file", name = "file", accept = ".csv"),
      tags$label(`for` = "file2", "Upload xlsx file"),
      tags$input(type = "file", id = "file2", name = "file2", accept = ".xlsx"),
      tags$button(type = "submit", "Submit")
    )
  }
}
```

```

    )

    tagList(form1, form2)
  }

home_get <- function(req, res) {
  html <- tagList(
    page_links(),
    tags$h3("hello, world!"),
    forms()
  )

  res$send(html)
}

home_post <- function(req, res) {
  body <- parse_json(req)
  # print(body)

  response <- list(
    code = 200L,
    msg = "hello, world"
  )
  res$json(response)
}

url_form_encoded_post <- function(req, res) {
  body <- parse_form_urlencoded(req)
  # print(body)

  list_items <- lapply(
    X = names(body),
    FUN = function(nm) {
      tags$li(
        nm,
        ":",
        body[[nm]]
      )
    }
  )
  input_vals <- tags$ul(list_items)

  html <- tagList(
    page_links(),
    tags$h3("Request processed"),
    input_vals
  )

  res$send(html)
}

multipart_form_data_post <- function(req, res) {
  body <- parse_multipart(req)

```

```

list_items <- lapply(
  X = names(body),
  FUN = function(nm) {
    field <- body[[nm]]

    # if 'field' is a file, parse it & print on console:
    is_file <- "filename" %in% names(field)
    is_csv <- is_file && identical(field[["content_type"]], "text/csv")
    is_xlsx <- is_file &&
      identical(
        field[["content_type"]],
        "application/vnd.openxmlformats-officedocument.spreadsheetml.sheet"
      )

    if (is_file) {
      file_path <- tempfile()
      writeBin(object = field$value, con = file_path)
      on.exit(unlink(x = file_path))
    }

    if (is_csv) {
      # print(read.csv(file = file_path))
    }

    if (is_xlsx) {
      # print(readxl::read_xlsx(path = file_path))
    }

    tags$li(
      nm,
      ":",
      if (is_file) "printed on console" else field
    )
  }
)
input_vals <- tags$ul(list_items)

html <- tagList(
  page_links(),
  tags$h3("Request processed"),
  input_vals
)

res$send(html)
}

about_get <- function(req, res) {
  html <- tagList(
    page_links(),
    tags$h3("About Us")
  )
  res$send(html)
}

```

```

    }

    contact_get <- function(req, res) {
      html <- tagList(
        page_links(),
        tags$h3("Get In Touch!")
      )
      res$send(html)
    }

    app <- Ambiorix$new(port = 5000L)

    app$
      get("/", home_get)$
      post("/", home_post)$
      get("/about", about_get)$
      get("/contact", contact_get)$
      post("/url-form-encoded", url_form_encoded_post)$
      post("/multipart-form-data", multipart_form_data_post)

    app$start()
  }

```

 parse_multipart

Parse multipart form data

Description

Parses multipart form data, including file uploads, and returns the parsed fields as a list.

Usage

```
parse_multipart(req, ...)
```

Arguments

req	The request object.
...	Additional parameters passed to the parser function.

Details

If a field is a file upload it is returned as a named list with:

- value: Raw vector representing the file contents. You must process this further (eg. convert to data.frame). See the examples section.
- content_disposition: Typically "form-data", indicating how the content is meant to be handled.
- content_type: MIME type of the uploaded file (e.g., "image/png" or "application/pdf").

- name: Name of the form input field.
- filename: Original name of the uploaded file.

If no body data, an empty list is returned.

Overriding Default Parser:

By default, `parse_multipart()` uses `webutils::parse_http()` internally. You can override this globally by setting the `AMBIORIX_MULTIPART_FORM_DATA_PARSER` option:

```
options(AMBIORIX_MULTIPART_FORM_DATA_PARSER = my_custom_parser)
```

Your custom parser function must accept the following parameters:

1. body: Raw vector containing the form data.
2. content_type: The 'Content-Type' header of the request as defined by the client.
3. ...: Additional optional parameters.

Value

Named list.

See Also

[parse_form_urlencoded\(\)](#), [parse_json\(\)](#)

Examples

```
if (interactive()) {
  library(ambiorix)
  library(htmltools)
  library(readxl)

  page_links <- function() {
    Map(
      f = function(href, label) {
        tags$a(href = href, label)
      },
      c("/", "/about", "/contact"),
      c("Home", "About", "Contact")
    )
  }

  forms <- function() {
    form1 <- tags$form(
      action = "/url-form-encoded",
      method = "POST",
      enctype = "application/x-www-form-urlencoded",
      tags$h4("form-url-encoded:"),
      tags$label(`for` = "first_name", "First Name"),
      tags$input(id = "first_name", name = "first_name", value = "John"),
      tags$label(`for` = "last_name", "Last Name"),
      tags$input(id = "last_name", name = "last_name", value = "Coene"),
      tags$button(type = "submit", "Submit")
    )
  }
}
```

```

)

form2 <- tags$form(
  action = "/multipart-form-data",
  method = "POST",
  enctype = "multipart/form-data",
  tags$h4("multipart/form-data:"),
  tags$label(`for` = "email", "Email"),
  tags$input(id = "email", name = "email", value = "john@mail.com"),
  tags$label(`for` = "framework", "Framework"),
  tags$input(id = "framework", name = "framework", value = "ambiorix"),
  tags$label(`for` = "file", "Upload CSV file"),
  tags$input(type = "file", id = "file", name = "file", accept = ".csv"),
  tags$label(`for` = "file2", "Upload xlsx file"),
  tags$input(type = "file", id = "file2", name = "file2", accept = ".xlsx"),
  tags$button(type = "submit", "Submit")
)

tagList(form1, form2)
}

home_get <- function(req, res) {
  html <- tagList(
    page_links(),
    tags$h3("hello, world!"),
    forms()
  )

  res$send(html)
}

home_post <- function(req, res) {
  body <- parse_json(req)
  # print(body)

  response <- list(
    code = 200L,
    msg = "hello, world"
  )
  res$json(response)
}

url_form_encoded_post <- function(req, res) {
  body <- parse_form_urlencoded(req)
  # print(body)

  list_items <- lapply(
    X = names(body),
    FUN = function(nm) {
      tags$li(
        nm,
        ":",
        body[[nm]]
      )
    }
  )
}

```

```

    )
  }
)
input_vals <- tags$ul(list_items)

html <- tagList(
  page_links(),
  tags$h3("Request processed"),
  input_vals
)

res$send(html)
}

multipart_form_data_post <- function(req, res) {
  body <- parse_multipart(req)

  list_items <- lapply(
    X = names(body),
    FUN = function(nm) {
      field <- body[[nm]]

      # if 'field' is a file, parse it & print on console:
      is_file <- "filename" %in% names(field)
      is_csv <- is_file && identical(field[["content_type"]], "text/csv")
      is_xlsx <- is_file &&
        identical(
          field[["content_type"]],
          "application/vnd.openxmlformats-officedocument.spreadsheetml.sheet"
        )

      if (is_file) {
        file_path <- tempfile()
        writeBin(object = field$value, con = file_path)
        on.exit(unlink(x = file_path))
      }

      if (is_csv) {
        # print(read.csv(file = file_path))
      }

      if (is_xlsx) {
        # print(readxl::read_xlsx(path = file_path))
      }

      tags$li(
        nm,
        ":",
        if (is_file) "printed on console" else field
      )
    }
  )
  input_vals <- tags$ul(list_items)
}

```

```

    html <- tagList(
      page_links(),
      tags$h3("Request processed"),
      input_vals
    )

    res$send(html)
  }

  about_get <- function(req, res) {
    html <- tagList(
      page_links(),
      tags$h3("About Us")
    )
    res$send(html)
  }

  contact_get <- function(req, res) {
    html <- tagList(
      page_links(),
      tags$h3("Get In Touch!")
    )
    res$send(html)
  }

  app <- Ambiorix$new(port = 5000L)

  app$
    get("/", home_get)$
    post("/", home_post)$
    get("/about", about_get)$
    get("/contact", contact_get)$
    post("/url-form-encoded", url_form_encoded_post)$
    post("/multipart-form-data", multipart_form_data_post)

  app$start()
}

```

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.

Value

A response pre-hook.

Examples

```
my_prh <- function(self, content, data, ext, ...) {
  data$title <- "Mansion"
  pre_hook(content, data)
}

#' Handler for GET at '/'
#'
#' @details Renders the homepage
#' @export
home_get <- function(req, res) {
  res$pre_render_hook(my_prh)
  res$render(
    file = "page.html",
    data = list(
      title = "Home"
    )
  )
}
```

Request	<i>Request</i>
---------	----------------

Description

A request.

Value

A Request object.

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.

SERVER_NAME Name of the server.

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". # @field 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\\$get_header\(\)](#)
- [Request\\$parse_multipart\(\)](#)
- [Request\\$parse_json\(\)](#)
- [Request\\$clone\(\)](#)

Request\$new():

Usage:

`Request$new(req)`

Arguments:

`req` Original request (environment).

Details: Constructor

Request\$print():

Usage:

`Request$print()`

Details: Print

Request\$get_header():

Usage:

`Request$get_header(name)`

Arguments:

`name` Name of the header

Details: Get Header

Request\$parse_multipart():

Usage:

`Request$parse_multipart()`

Details: Parse Multipart encoded data

Request\$parse_json():

Usage:

`Request$parse_json(...)`

Arguments:

`...` Arguments passed to [parse_json\(\)](#).

Details: Parse JSON encoded data

Request\$clone(): The objects of this class are cloneable with this method.

Usage:

`Request$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

Examples

```

if (interactive()) {
  library(ambiorix)

  app <- Ambiorix$new()

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

  app$start()
}

```

Response	<i>Response</i>
----------	-----------------

Description

Response class to generate responses sent from the server.

Value

A Response object.

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$print()`
- `Response$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$pre_render_hook()`
- `Response$post_render_hook()`
- `Response$cookie()`
- `Response$clear_cookie()`
- `Response$clone()`

`Response$set_status()`:

Usage:

`Response$set_status(status)`

Arguments:

`status` An integer defining the status.

Details: Set the status of the response.

`Response$send()`:

Usage:

`Response$send(body)`

Arguments:

`body` Body of the response.

Details: Send a plain HTML response.

`Response$sendf()`:

Usage:

`Response$sendf(body, ...)`

Arguments:

`body` Body of the response.

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

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

`Response$text():`

Usage:

`Response$text(body)`

Arguments:

`body` Body of the response.

Details: Send a plain text response.

`Response$send_file():`

Usage:

`Response$send_file(file)`

Arguments:

`file` File to send.

Details: Send a file.

`Response$redirect():`

Usage:

`Response$redirect(path)`

Arguments:

`path` Path or URL to redirect to.

Details: Redirect to a path or URL.

`Response$render():`

Usage:

`Response$render(file, data = list())`

Arguments:

`file` Template file.

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

Details: Render a template file.

`Response$json():`

Usage:

`Response$json(body, ...)`

Arguments:

`body` Body of the response.

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

Details: Render an object as JSON.

`Response$csv():`

Usage:

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

Arguments:

`data` Data to convert to CSV.
`name` Name of the file.
... Additional arguments passed to `readr::format_csv()`.

Details: Sends a comma separated value file

`Response$tsv()`:

Usage:

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

Arguments:

`data` Data to convert to CSV.
`name` Name of the file.
... Additional arguments passed to `readr::format_tsv()`.

Details: Sends a tab separated value file

`Response$htmlwidget()`:

Usage:

`Response$htmlwidget(widget, ...)`

Arguments:

`widget` The widget to use.
... Additional arguments passed to `htmlwidgets::saveWidget()`.

Details: Sends an htmlwidget.

`Response$md()`:

Usage:

`Response$md(file, data = list())`

Arguments:

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

Details: Render a markdown file.

`Response$png()`:

Usage:

`Response$png(file)`

Arguments:

`file` Path to local file.

Details: Send a png file

`Response$jpeg()`:

Usage:

`Response$jpeg(file)`

Arguments:

file Path to local file.

Details: Send a jpeg file

`Response$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.

`Response$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

`Response$print():`

Usage:

`Response$print()`

Details: Print

`Response$header():`

Usage:

`Response$header(name, value)`

Arguments:

name String. Name of the header.

value Value of the header.

Details: Add headers to the response.

Returns: Invisibly returns self.

`Response$header_content_json():`

Usage:

`Response$header_content_json()`

Details: Set Content Type to JSON

Returns: Invisibly returns self.

`Response$header_content_html():`

Usage:

Response\$header_content_html()

Details: Set Content Type to HTML

Returns: Invisibly returns self.

Response\$header_content_plain():

Usage:

Response\$header_content_plain()

Details: Set Content Type to Plain Text

Returns: Invisibly returns self.

Response\$header_content_csv():

Usage:

Response\$header_content_csv()

Details: Set Content Type to CSV

Returns: Invisibly returns self.

Response\$header_content_tsv():

Usage:

Response\$header_content_tsv()

Details: Set Content Type to TSV

Returns: Invisibly returns self.

Response\$get_headers():

Usage:

Response\$get_headers()

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

Response\$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.

Response\$set_headers():

Usage:

Response\$set_headers(headers)

Arguments:

headers A named list of headers to set.

Details: Set headers

`Response$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 method.

Returns: Invisible returns self.

`Response$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 Response 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.

`Response$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.samesite")
)
```

Arguments:

`name` String. 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.

`Response$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.

`Response$clone():` The objects of this class are cloneable with this method.

Usage:

`Response$clone(deep = FALSE)`

Arguments:

deep Whether to make a deep clone.

Examples

```
if (interactive()) {
  library(ambiorix)

  app <- Ambiorix$new()

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

```
  })  
  
  app$start()  
}
```

responses	<i>Plain Responses</i>
-----------	------------------------

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

Value

An Ambiorix response.

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()
```

roby

R Object

Description

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

Usage

```
roby(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 replaced like `x <- hello` (missing quote): breaking the template. Using `roby` one would obtain `x <- "hello"`.

Value

Object of class "roby".

Examples

```
roby(1:10)
```

Router

Router

Description

Web server.

Value

A Router object.

Super class

[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\(\)](#)

Router\$new():

Usage:

Router\$new(path)

Arguments:

path The base path of the router.

Details: Define the base route.

Router\$print():

Usage:

Router\$print()

Details: Print

Router\$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](#).

Value

A Routing object.

HTTP methods

See [routing-http-methods](#) for the full argument reference. The routing instance exposes helpers for common HTTP verbs; they are registered when the object is initialised and share the same signature.

- `get()`, `put()`, `patch()`, `delete()`, `post()`, `options()` register a handler for the single corresponding HTTP verb; see [routing-http-methods](#).
- `all()` registers a handler that responds to GET, POST, PUT, DELETE, and PATCH; see [routing-http-methods](#).

Public fields

`error` Error handler.

`get` Register a route handler for HTTP GET requests. See [routing-http-methods](#).

`put` Register a route handler for HTTP PUT requests. See [routing-http-methods](#).

`patch` Register a route handler for HTTP PATCH requests. See [routing-http-methods](#).

`delete` Register a route handler for HTTP DELETE requests. See [routing-http-methods](#).

`post` Register a route handler for HTTP POST requests. See [routing-http-methods](#).

`options` Register a route handler for HTTP OPTIONS requests. See [routing-http-methods](#).

`all` Register a route handler that responds to every HTTP verb used by Ambiorix. See [routing-http-methods](#).

Active bindings

`basepath` Basepath, read-only.

`websocket` Websocket handler.

Methods

Public methods:

- [Routing\\$new\(\)](#)
- [Routing\\$param\(\)](#)
- [Routing\\$receive\(\)](#)
- [Routing\\$print\(\)](#)
- [Routing\\$engine\(\)](#)
- [Routing\\$use\(\)](#)
- [Routing\\$get_routes\(\)](#)
- [Routing\\$get_params\(\)](#)
- [Routing\\$get_receivers\(\)](#)
- [Routing\\$get_middleware\(\)](#)
- [Routing\\$prepare\(\)](#)
- [Routing\\$clone\(\)](#)

`Routing$new():`

Usage:

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

Arguments:

`path` Prefix path.

Details: Initialise

Routing\$param():

Usage:

Routing\$param(name, handler)

Arguments:

name Name of the parameter

handler Function that accepts the request, response, parameter value and the parameter name.

Details: PARAM Method

Examples:

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

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

```
app$param("person", function(req, res, value, name){  
  if(value == "notWanted"){  
    res$status <- 403L  
    res$send("This is the end.")  
  }  
}
```

```
  # continue processing the request...  
})
```

```
app$get("/hi/:person", function(req,res){  
  res$sendf("Hi! %s", req$params$person)  
})  
app$get("/info/:person", function(req,res){  
  res$sendf("Here is all your info, %s", req$params$person)  
})  
if(interactive())  
  app$start()
```

Routing\$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()

```

Routing\$print():

Usage:

Routing\$print()

Details: Print

Routing\$engine():

Usage:

Routing\$engine(engine)

Arguments:

engine Engine function.

Details: Engine to use for rendering templates.

Routing\$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

Routing\$get_routes():

Usage:

Routing\$get_routes(routes = list(), parent = "")

Arguments:

routes Existing list of routes.

parent Parent path.

Details: Get the routes

`Routing$get_params():`

Usage:

`Routing$get_params(params = list(), parent = "")`

Arguments:

`params` Existing list of parameter middlewares.

`parent` Parent path.

Details: Get the parameter middlewares

`Routing$get_receivers():`

Usage:

`Routing$get_receivers(receivers = list())`

Arguments:

`receivers` Existing list of receivers

Details: Get the websocket receivers

`Routing$get_middleware():`

Usage:

`Routing$get_middleware(middlewares = list(), parent = "")`

Arguments:

`middlewares` Existing list of middlewares

`parent` Parent path

Details: Get the middleware

`Routing$prepare():`

Usage:

`Routing$prepare()`

Details: Prepare routes and decomposes paths

`Routing$clone():` The objects of this class are cloneable with this method.

Usage:

`Routing$clone(deep = FALSE)`

Arguments:

`deep` Whether to make a deep clone.

See Also

[routing-http-methods](#)

Examples

```
## -----
## Method `Routing$param()`
## -----

app <- Ambiorix$new()

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

app$param("person", function(req, res, value, name){
  if(value == "notWanted"){
    res$status <- 403L
    res$send("This is the end.")
  }

  # continue processing the request...
})

app$get("/hi/:person", function(req,res){
  res$sendf("Hi! %s", req$params$person)
})
app$get("/info/:person", function(req,res){
  res$sendf("Here is all your info, %s", req$params$person)
})
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()
```

Description

Register route handlers for HTTP verbs on a [Routing](#) instance.

Arguments

path	String. Route to listen to, treated as a regular expression; use <code>:</code> to define a parameter (e.g. <code>"/hello/:name"</code>). See the <i>Path matching</i> section.
handler	Function that accepts the request and response objects and returns an httpuv response (e.g. response()). Handlers can return the result of helper functions such as <code>Response\$text()</code> , <code>Response\$json()</code> , or the output of any renderer.
error	Optional handler invoked if the route raises an error; receives the request, response, and the error condition.
docs	Optional OpenAPI documentation for the route, created with openapi_docs() . When the app enables docs via <code>app\$openapi()</code> , documented routes appear in the generated OpenAPI document. Path parameters are documented automatically from the route's <code>:param</code> tokens with a string schema; declare them via openapi_param() with <code>location = "path"</code> to override that default.

Details

The routing helpers provide a fluent API for attaching handlers to HTTP methods. Each helper shares the same signature and behaviour.

Supported helpers:

- `get()`: Respond to HTTP GET requests.
- `post()`: Respond to HTTP POST requests.
- `put()`: Respond to HTTP PUT requests.
- `patch()`: Respond to HTTP PATCH requests.
- `delete()`: Respond to HTTP DELETE requests.
- `options()`: Respond to HTTP OPTIONS requests.
- `all()`: Respond to every method above.

Path matching:

Paths are treated as regular expressions; use `:` to define a parameter (e.g. `"/hello/:name"`).

- Parameters match greedily, across `/`: `"/users/:res"` matches `/users/1` as well as `/users/2/3`. Note that `req$params` captures a single path segment, so for `/users/2/3` the value of `req$params$res` is `"2"`.
- Regular expression syntax is available for finer control, e.g. `app$get("/users/.+", ...)` for a greedy match without a parameter, or `app$get("/file\\.json", ...)` to match a literal dot (an unescaped `.` matches any character).
- To customise how paths are converted to patterns app-wide, see [as_path_to_pattern\(\)](#).

Value

The routing object invisibly so calls can be chained.

See Also

[Routing](#), [openapi_docs\(\)](#)

Examples

```
app <- Ambiorix$new()

app$get("/", function(req, res) {
  res$text("Hello, world!")
})

app$post("/echo", function(req, res) {
  res$json(list(received = req$body))
})

app$all("/health", function(req, res) {
  res$json(list(status = "ok"))
})

app$get(
  "/users/:id",
  function(req, res) {
    res$json(list(id = req$params$id))
  },
  docs = openapi_docs(
    summary = "Get a user by ID",
    tags = "users",
    responses = openapi_responses(
      openapi_response(200, "The user")
    )
  )
)
```

serialise

Serialise an Object to JSON

Description

Serialise an Object to JSON

Usage

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

Arguments

<code>data</code>	Data to serialise.
<code>...</code>	Passed to serialiser.

Details

Ambiorix uses `yyjsonr::write_json_str()` by default for serialization.

Custom Serialiser:

To override the default, set the `AMBIORIX_SERIALISER` option to a function that accepts:

- `data`: Object to serialise.
- `...`: Additional arguments passed to the function.

For example:

```
my_serialiser <- function(data, ...) {  
  jsonlite::toJSON(x = data, ...)  
}  
  
options(AMBIORIX_SERIALISER = my_serialiser)
```

Value

JSON string.

Examples

```
if (interactive()) {  
  # a list:  
  response <- list(code = 200L, msg = "hello, world!")  
  
  serialise(response)  
  #> {"code":200,"msg":"hello, world"}  
  
  serialise(response, auto_unbox = FALSE)  
  #> {"code":[200],"msg":["hello, world"]}  
  
  # data.frame:  
  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](#).

Value

The log object.

Examples

```
# define custom loggers:
info_logger <- log::Logger$new("INFO")
success_logger <- log::Logger$new("SUCCESS")
error_logger <- log::Logger$new("ERROR")

info_logger$log("This is an info message.")
success_logger$log("This is a success message.")
error_logger$log("This is an error message.")

# set custom loggers for Ambiorix:
set_log_info(info_logger)
set_log_success(success_logger)
set_log_error(error_logger)
```

stop_all

Stop

Description

Stop all servers.

Usage

```
stop_all()
```

Value

NULL (invisibly)

Examples

```
if (interactive()) {
  stop_all()
}
```

token_create	<i>Token</i>
--------------	--------------

Description

Create a token

Usage

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

Arguments

n Number of bytes.

Value

Length 1 character vector.

Examples

```
token_create()  
token_create(n = 32L)
```

use_html_template	<i>HTML Template</i>
-------------------	----------------------

Description

Use `htmltools::htmlTemplate()` as renderer. Passed to use method.

Usage

```
use_html_template()
```

Value

A renderer function.

Examples

```
use_html_template()
```

Websocket

Websocket

Description

Handle websocket messages.

Value

A Websocket object.

Methods**Public methods:**

- [Websocket\\$new\(\)](#)
- [Websocket\\$send\(\)](#)
- [Websocket\\$print\(\)](#)
- [Websocket\\$clone\(\)](#)

Websocket\$new():

Usage:

Websocket\$new(ws)

Arguments:

ws The websocket

Details: Constructor

Websocket\$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

Websocket\$print():

Usage:

Websocket\$print()

Details: Print

Websocket\$clone(): The objects of this class are cloneable with this method.

Usage:

Websocket\$clone(deep = FALSE)

Arguments:

deep Whether to make a deep clone.

Examples

```
# create an Ambiorix app with websocket support:
if (interactive()) {
  library(ambiorix)

  home_get <- function(req, res) {
    res$send("hello, world!")
  }

  greeting_ws_handler <- function(msg, ws) {
    cat("Received message:", "\n")
    print(msg)
    ws$send("greeting", "Hello from the server!")
  }

  app <- Ambiorix$new(port = 8080)
  app$get("/", home_get)
  app$receive("greeting", greeting_ws_handler)
  app$start()
}

# create websocket client from another R session:
if (interactive()) {
  client <- websocket::WebSocket$new("ws://127.0.0.1:8080", autoConnect = FALSE)

  client$onOpen(function(event) {
    cat("Connection opened\n")

    msg <- list(
      isAmbiorix = TRUE, # __MUST__ be set!
      name = "greeting",
      message = "Hello from the client!"
    )

    # serialise:
    msg <- yyjsonr::write_json_str(msg, auto_unbox = TRUE)

    client$send(msg)
  })

  client$onMessage(function(event) {
    cat("Received message from server:", event$data, "\n")
  })

  client$connect()
}
```

Description

Handle ambiorix websocket client.

Usage

```
copy_websocket_client(path)
```

```
get_websocket_client_path()
```

```
get_websocket_clients()
```

Arguments

path Path to copy the file to.

Value

- `copy_websocket_client`: String. The new path (invisibly).
- `get_websocket_client_path`: String. The path to the local websocket client.
- `get_websocket_clients`: List. Websocket clients.

Functions

- `copy_websocket_client` Copies the websocket client file, useful when ambiorix was not setup with the ambiorix generator.
- `get_websocket_client_path` Retrieves the full path to the local websocket client.
- `get_websocket_clients` Retrieves clients connected to the server.

Examples

```
chat_ws <- function(msg, ws) {  
  lapply(  
    X = get_websocket_clients(),  
    FUN = function(c) {  
      c$send("chat", msg)  
    }  
  )  
}
```

Index

* export

- Routing, [51](#)
- Ambiorix, [2](#), [51](#)
- as_cookie_parser, [10](#)
- as_cookie_preprocessor, [11](#)
- as_path_to_pattern, [12](#)
- as_path_to_pattern(), [57](#)
- as_renderer, [13](#)
- content, [13](#)
- content_csv (content), [13](#)
- content_html (content), [13](#)
- content_json (content), [13](#)
- content_plain (content), [13](#)
- content_protobuf (content), [13](#)
- content_tsv (content), [13](#)
- copy_websocket_client
 (websocket_client), [63](#)
- default_cookie_parser, [14](#)
- dput(), [49](#)
- forward, [15](#)
- get_websocket_client_path
 (websocket_client), [63](#)
- get_websocket_clients
 (websocket_client), [63](#)
- ggplot2::ggsave(), [44](#)
- htmltools::htmlTemplate(), [61](#)
- htmlwidgets::saveWidget(), [43](#)
- import, [16](#)
- jobj, [16](#)
- log::Logger, [60](#)
- mockRequest, [17](#)
- new_log, [17](#)
- openapi-schemas, [18](#), [20](#), [22](#), [23](#)
- openapi_docs, [19](#)
- openapi_docs(), [7](#), [57](#), [58](#)
- openapi_param, [20](#)
- openapi_param(), [19](#), [21](#), [57](#)
- openapi_parameters, [21](#)
- openapi_parameters(), [19](#)
- openapi_request_body, [22](#)
- openapi_request_body(), [19](#)
- openapi_response, [22](#)
- openapi_response(), [23](#)
- openapi_responses, [23](#)
- openapi_responses(), [19](#)
- openapi_schema_array (openapi-schemas),
 [18](#)
- openapi_schema_boolean
 (openapi-schemas), [18](#)
- openapi_schema_integer
 (openapi-schemas), [18](#)
- openapi_schema_number
 (openapi-schemas), [18](#)
- openapi_schema_object
 (openapi-schemas), [18](#)
- openapi_schema_string
 (openapi-schemas), [18](#)
- parse_form_urlencoded, [24](#)
- parse_form_urlencoded(), [29](#), [33](#)
- parse_json, [28](#)
- parse_json(), [24](#), [33](#), [39](#)
- parse_multipart, [32](#)
- parse_multipart(), [24](#), [29](#)
- pre_hook, [36](#)
- pre_hook(), [46](#)
- readr::format_csv(), [43](#)
- readr::format_tsv(), [43](#)
- Request, [11](#), [14](#), [37](#)

Response, [11](#), [40](#)
response (responses), [48](#)
response(), [4](#), [57](#)
response_404 (responses), [48](#)
response_404(), [3](#)
response_500 (responses), [48](#)
response_500(), [3](#), [50](#)
responses, [48](#)
robject, [49](#)
Router, [50](#), [51](#), [54](#)
Routing, [3](#), [50](#), [51](#), [57](#), [58](#)
routing-http-methods, [19](#), [20](#), [51](#), [52](#), [55](#), [56](#)

serialise, [58](#)
set_log, [59](#)
set_log_error (set_log), [59](#)
set_log_info (set_log), [59](#)
set_log_success (set_log), [59](#)
stop_all, [60](#)

token_create, [61](#)

use_html_template, [61](#)

Websocket, [62](#)
websocket_client, [63](#)
webutils::parse_http(), [24](#), [33](#)

yyjsonr::read_json_raw(), [28](#)
yyjsonr::write_json_str(), [59](#)