Package 'RestRserve'

November 11, 2020

Type Package

Title A Framework for Building HTTP API

Description Allows to easily create high-performance full featured HTTP APIs from R functions. Provides high-level classes such as 'Request', 'Response', 'Application', 'Middleware' in order to streamline server side application development. Out of the box allows to serve requests using 'Rserve' package, but flexible enough to integrate with other HTTP servers such as 'httpuv'.

Version 0.4.0

URL https://restrserve.org, https://github.com/rexyai/RestRserve

BugReports https://github.com/rexyai/RestRserve/issues

License GPL (>= 2)

Depends R (>= 3.6.0)

Imports methods, parallel, Rserve (>= 1.7.3), Rcpp (>= 1.0.3), R6 (>= 2.4.0), uuid (>= 0.1-2), checkmate (>= 1.9.4), mime (>= 0.7), jsonlite (>= 1.6)

Suggests tinytest (>= 1.0.0), lgr (>= 0.3.2), lintr, knitr, rmarkdown, curl, sys

LinkingTo Rcpp

SystemRequirements C++11

ByteCompile true

KeepSource true

Encoding UTF-8

LazyData true

RoxygenNote 7.1.1

VignetteBuilder knitr

NeedsCompilation yes

Author Dmitriy Selivanov [aut, cre] (<https://orcid.org/0000-0001-5413-1506>), Artem Klevtsov [aut] (<https://orcid.org/0000-0003-0492-6647>), rexy.ai [cph, fnd]

Maintainer Dmitriy Selivanov <ds@rexy.ai>

Repository CRAN

Date/Publication 2020-11-11 10:00:12 UTC

R topics documented:

`Application` *Creates application - RestRserve usage starts from here*

Description

Creates Application object. Application provides an interface for building high-performance REST API by registering R functions as handlers http requests.

Public fields

`logger` Logger object which records events during request processing. Alternatively user can use loggers from lgr package as a drop-in replacement - `Logger` methods and loggers created by `lgr` share function signatures.

`content_type` Default response body content type.

`HTTPError` Class which raises HTTP errors. Global HTTPError is used by default. In theory user can replace it with his own class (see `RestRserve:::HTTPErrorFactory`). However we believe in the majority of the cases using HTTPError will be enough.

Active bindings

`endpoints` Prints all the registered routes with allowed methods.

Methods

Public methods:

- `Application$new()`
- `Application$add_route()`
- `Application$add_get()`
- `Application$add_post()`
- `Application$add_static()`
- `Application$add_openapi()`
- `Application$add_swagger_ui()`
- `Application$append_middleware()`
- `Application$process_request()`
- `Application$print()`
- `Application$clone()`

Method `new()`**:** Creates Application object.

Usage:

```
Application$new(
  middleware = list(EncodeDecodeMiddleware$new()),
  content_type = "text/plain",
  ...
)
```

Arguments:

`middleware` List of Middleware objects.

`content_type` Default response body content (media) type. `"text/plain"` by default.

`...` Not used at the moment.

Method `add_route()`**:** Adds endpoint and register user-supplied R function as a handler.

Usage:

```
Application$add_route(
  path,
  method,
  FUN,
  match = c("exact", "partial", "regex"),
  ...
)
```

Arguments:

`path` Endpoint path.

`method` HTTP method. Allowed methods at the moment: `GET`, `HEAD`, `POST`, `PUT`, `DELETE`, `OPTIONS`, `PATCH`.

`FUN` User function to handle requests. `FUN` **must** take two arguments: first is `request` (Request) and second is `response` (Response).
The goal of the user function is to **modify** `response` or throw exception (call `raise()` or `stop()`).
Both `response` and `request` objects modified in-place and internally passed further to RestRserve execution pipeline.

`match` Defines how route will be processed. Allowed values:

- `exact` - match route as is. Returns 404 if route is not matched.
- `partial` - match route as prefix. Returns 404 if prefix are not matched.
- `regex` - match route as template. Returns 404 if template pattern not matched.

`...` Not used.

Method `add_get()`**:** Shorthand to `Application$add_route()` with `GET` method.

Usage:

```
Application$add_get(
  path,
  FUN,
  match = c("exact", "partial", "regex"),
  ...,
  add_head = TRUE
)
```

Arguments:

`path` Endpoint path.

`FUN` User function to handle requests. `FUN` **must** take two arguments: first is `request` (Request) and second is `response` (Response).
The goal of the user function is to **modify** `response` or throw exception (call `raise()` or `stop()`).
Both `response` and `request` objects modified in-place and internally passed further to RestRserve execution pipeline.

`match` Defines how route will be processed. Allowed values:

- `exact` - match route as is. Returns 404 if route is not matched.
- `partial` - match route as prefix. Returns 404 if prefix are not matched.
- `regex` - match route as template. Returns 404 if template pattern not matched.

`...` Not used.

`add_head` Adds HEAD method.

Method `add_post()`**:** Shorthand to `Application$add_route()` with `POST` method.

Usage:

```
Application$add_post(path, FUN, match = c("exact", "partial", "regex"), ...)
```

Arguments:

`path` Endpoint path.

`FUN` User function to handle requests. `FUN` **must** take two arguments: first is `request` (Request) and second is `response` (Response).
The goal of the user function is to **modify** `response` or throw exception (call `raise()` or `stop()`).
Both `response` and `request` objects modified in-place and internally passed further to RestRserve execution pipeline.

`match` Defines how route will be processed. Allowed values:

- `exact` - match route as is. Returns 404 if route is not matched.
- `partial` - match route as prefix. Returns 404 if prefix are not matched.

- `regex` - match route as template. Returns 404 if template pattern not matched.

`...` Not used.

Method `add_static()`**:** Adds `GET` method to serve file or directory at `file_path`.

Usage:

```
Application$add_static(path, file_path, content_type = NULL, ...)
```

Arguments:

`path` Endpoint path.

`file_path` Path file or directory.

`content_type` MIME-type for the content.
If `content_type = NULL` then MIME code `content_type` will be inferred automatically (from file extension).
If it will be impossible to guess about file type then `content_type` will be set to `application/octet-stream`.

`...` Not used.

Method `add_openapi()`**:** Adds endpoint to serve OpenAPI description of available methods.

Usage:

```
Application$add_openapi(path = "/openapi.yaml", file_path = "openapi.yaml")
```

Arguments:

`path` path Endpoint path.

`file_path` Path to the OpenAPI specification file.

Method `add_swagger_ui()`**:** Adds endpoint to show Swagger UI.

Usage:

```
Application$add_swagger_ui(
  path = "/swagger",
  path_openapi = "/openapi.yaml",
  use_cdn = TRUE,
  path_swagger_assets = "/__swagger__/",
  file_path = "swagger-ui.html"
)
```

Arguments:

`path` path Endpoint path.

`path_openapi` Path to the OpenAPI specification file.

`use_cdn` Use CDN to load Swagger UI libraries.

`path_swagger_assets` Swagger UI asstes endpoint.

`file_path` Path to Swagger UI HTML file.

Method `append_middleware()`**:** Appends middleware to handlers pipeline.

Usage:

```
Application$append_middleware(mw)
```

Arguments:

`mw` Middleware object.

Method `process_request()`**:** Process incoming request and generate Response object.

Usage:

```
Application$process_request(request = NULL)
```

Arguments:

`request` Request object.
Useful for tests your handlers before deploy application.

Method `print()`**:** Prints application details.

Usage:

```
Application$print()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

HTTPError Middleware Request Response

Examples

```
# init logger
app_logger = Logger$new()
# set log level for the middleware
app_logger$set_log_level("debug")
# set logger name
app_logger$set_name("MW Logger")
# init middleware to logging
mw = Middleware$new(
  process_request = function(rq, rs) {
    app_logger$info(sprintf("Incomming request (id %s): %s", rq$id, rq$path))
  },
  process_response = function(rq, rs) {
    app_logger$info(sprintf("Outgoing response (id %s): %s", rq$id, rs$status))
  },
  id = "awesome-app-logger"
)

# init application
app = Application$new(middleware = list(mw))

# set internal log level
app$logger$set_log_level("error")

# define simply request handler
status_handler = function(rq, rs) {
  rs$set_body("OK")
```

```
  rs$set_content_type("text/plain")
  rs$set_status_code(200L)
}
# add route
app$add_get("/status", status_handler, "exact")

# add static file handler
desc_file = system.file("DESCRIPTION", package = "RestRserve")
# add route
app$add_static("/desc", desc_file, "text/plain")

# define say message handler
say_handler = function(rq, rs) {
  who = rq$parameters_path[["user"]]
  msg = rq$parameters_query[["message"]]
  if (is.null(msg)) msg <- "Hello"
  rs$set_body(paste(who, "say", dQuote(msg)))
  rs$set_content_type("text/plain")
  rs$set_status_code(200L)
}
# add route
app$add_get("/say/{user}", say_handler, "regex")

# print application info
app

# test app
# simulate requests
not_found_rq = Request$new(path = "/no")
status_rq = Request$new(path = "/status")
desc_rq = Request$new(path = "/desc")
say_rq = Request$new(path = "/say/anonym", parameters_query = list("message" = "Hola"))
# process prepared requests
app$process_request(not_found_rq)
app$process_request(status_rq)
app$process_request(desc_rq)
app$process_request(say_rq)

# run app
backend = BackendRserve$new()

if (interactive()) {
  backend$start(app, 8080)
}
```

`ApplicationProcess` *Creates ApplicationProcess object*

Description

Creates ApplicationProcess to hold PID of the runnung applicaiton.

Public fields

`pid` Process identificator.

Methods

Public methods:

- `ApplicationProcess$new()`
- `ApplicationProcess$kill()`
- `ApplicationProcess$clone()`

Method `new()`**:** Creates ApplicationProcess object

Usage:

```
ApplicationProcess$new(pid)
```

Arguments:

`pid` Process identificator.

Method `kill()`**:** Send signal to process.

Usage:

```
ApplicationProcess$kill(signal = 15L)
```

Arguments:

`signal` Singal code.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`AuthBackendBasic` *Basic authorization backend*

Description

Creates AuthBackendBasic class object.

Super class

`RestRserve::AuthBackend` -> `AuthBackendBasic`

Methods

Public methods:

- AuthBackendBasic$new()
- AuthBackendBasic$authenticate()
- AuthBackendBasic$clone()

Method `new()`**:** Creates AuthBackendBasic class object.

Usage:

```
AuthBackendBasic$new(FUN)
```

Arguments:

`FUN` Function to perform authentication which takes two arguments - `user` and `password`. Returns boolean - whether access is allowed for a requested `user` or not.

Method `authenticate()`**:** Provide authentication for the given request.

Usage:

```
AuthBackendBasic$authenticate(request, response)
```

Arguments:

`request` Request object.

`response` Response object.

Returns: Boolean - whether access is allowed for a requested `user` or not.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

References

RFC7617 Wikipedia

See Also

AuthMiddleware Request Response

Other AuthBackend: `AuthBackendBearer`, `AuthBackend`, `AuthMiddleware`

Examples

```
# init users database
user_db = list(
  "user-1" = "password-1",
  "user-2" = "password-2"
)
# define authentication handler
auth_fun = function(user, password) {
```

```
  if (is.null(user_db[[user]])) return(FALSE) # not found
  if (!identical(user_db[[user]], password)) return(FALSE) # incorrect
  return(TRUE)
}
# init backend
auth_backend = AuthBackendBasic$new(FUN = auth_fun)

# test backend
# define credentials (see RFC)
creds = jsonlite::base64_enc("user-1:password-1")
# generate request headers
h = list("Authorization" = sprintf("Basic %s", creds))
# simulate request
rq = Request$new(path = "/", headers = h)
# init response object
rs = Response$new()
# perform authentication
auth_backend$authenticate(rq, rs) # TRUE
```

`AuthBackendBearer` *Bearer token authorization backend*

Description

Creates AuthBackendBearer class object.

Super class

`RestRserve::AuthBackend` -> `AuthBackendBearer`

Methods

Public methods:

- `AuthBackendBearer$new()`
- `AuthBackendBearer$authenticate()`
- `AuthBackendBearer$clone()`

Method `new()`**:** Creates AuthBackendBearer class object.

Usage:

```
AuthBackendBearer$new(FUN)
```

Arguments:

`FUN` Function to perform authentication which takes one arguments - `token`. Returns boolean - whether access is allowed for a requested `token` or not.

Method `authenticate()`**:** Provide authentication for the given request.

Usage:

```
AuthBackendBearer$authenticate(request, response)
```

Arguments:

`request` Request object.

`response` Response object.

Returns: Boolean - whether access is allowed for a requested `user` or not.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

References

RFC6750 Specification

See Also

AuthMiddleware Request Response

Other AuthBackend: `AuthBackendBasic`, `AuthBackend`, `AuthMiddleware`

Examples

```
token_db = list(
  "valid-token" = as.POSIXct("2099-12-31", tz = "GMT"),
  "expired-token" = as.POSIXct("1900-01-01", tz = "GMT")
)
auth_fun = function(token) {
  if (is.null(token_db[[token]])) return(FALSE) # not found
  if (Sys.time() > token_db[[token]]) return(FALSE) # expired
  return(TRUE)
}
# init backend
auth_backend = AuthBackendBearer$new(FUN = auth_fun)

# test backend
# define credentials (see RFC)
token = "valid-token"
# generate request headers
h = list("Authorization" = sprintf("Bearer %s", token))
# simulate request
rq = Request$new(path = "/", headers = h)
# init response object
rs = Response$new()
# perform authentication
auth_backend$authenticate(rq, rs) # TRUE
```

`AuthMiddleware` *Creates authorization middleware object*

Description

Adds various authorizations to Application.

Super class

`RestRserve::Middleware` -> `AuthMiddleware`

Methods

Public methods:

- `AuthMiddleware$new()`
- `AuthMiddleware$clone()`

Method `new()`**:** Creeates AuthMiddleware object.

Usage:

```
AuthMiddleware$new(
  auth_backend,
  routes,
  match = "exact",
  id = "AuthMiddleware"
)
```

Arguments:

`auth_backend` Authentication backend.

`routes` Routes paths to protect.

`match` How routes will be matched: `"exact"` or `"partial"` (as prefix).

`id` Middleware id.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Middleware Application

Other AuthBackend: `AuthBackendBasic`, `AuthBackendBearer`, `AuthBackend`

BackendRserve *Creates Rserve backend for processing HTTP requests*

Description

Creates BackendRserve object which can start Application using Rserve backend.

Super class

RestRserve::Backend -> BackendRserve

Methods

Public methods:

- BackendRserve$new()
- BackendRserve$start()
- BackendRserve$set_request()
- BackendRserve$convert_response()
- BackendRserve$clone()

Method `new()`**:** Creates BackendRserve object.

Usage:

```
BackendRserve$new(..., jit_level = 0L, precompile = FALSE)
```

Arguments:

`...` Not used at the moment.

`jit_level` changes R's byte compiler level to this value before app start.

`precompile` try to use R's byte compiler to pre-compile

Method `start()`**:** Starts RestRserve application from current R session.

Usage:

```
BackendRserve$start(app, http_port = 8080, ..., background = FALSE)
```

Arguments:

`app` Application object.

`http_port` HTTP port for application. Negative values (such as -1) means not to expose plain http.

`...` Key-value pairs of the Rserve configuration. If contains `"http.port"` then `http_port` will be silently replaced with its value.

`background` Whether to try to launch in background process on UNIX.

Returns: ApplicationProcess object when `background = TRUE`.

Method `set_request()`**:** Parse request and set to it fields.

Usage:

```
BackendRserve$set_request(
  request,
  path = "/",
  parameters_query = NULL,
  headers = NULL,
  body = NULL
)
```

Arguments:

`request` Request object.

`path` Character with requested path. Always starts with `/`.

`parameters_query` A named character vector with URL decoded query parameters.

`headers` Request HTTP headers.

`body` Request body. Can be `NULL`, raw vector or named character vector for the URL encoded form (like a `parameters_query` parameter).

Returns: `request` modified object.

Method `convert_response()`**:** Convert `self` object to Rserve compatible structure.

Usage:

```
BackendRserve$convert_response(response)
```

Arguments:

`response` Response object.

Returns: List with the following structure:

- `body`: can be a character vector of length one or a raw vector. if the character vector is named "file" then the content of a file of that name is the body. If the character vector is named "tmpfile" then the content of a temporary file of that name is the body.
- `content-type`: must be a character vector of length one or NULL (if present, else default is `"text/plain"`).
- `headers`: must be a character vector - the elements will have CRLF appended and neither `Content-type` nor `Content-length` may be used.
- `status-code`: must be an integer if present (default is 200).

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

References

See http.c in Rserve

`CORSMiddleware` *Creates CORS middleware object*

Description

Adds CORS to Application. CORS Middleware out of the box in RestRserve to turn on/off the CORS Headers on preflight validation from the browser.

Cross Origin Resource Sharing is an additional security check done by moderns browsers to avoid request between different domains. To allow it RestRserve has easy way to enable your CORS policies. By default CORS policies are disabled. So if any request is coming from a different domain will be blocked by the browser as default because RestRserve will not send the headers required by the browser to allow cross site resource sharing. You can change this easy just by providing `CORSMiddleware` as middleware to the Application.

Super class

`RestRserve::Middleware` -> `CORSMiddleware`

Methods

Public methods:

- `CORSMiddleware$new()`
- `CORSMiddleware$clone()`

Method `new()`**:** Creates CORS middleware object

Usage:

```
CORSMiddleware$new(routes = "/", match = "partial", id = "CORSMiddleware")
```

Arguments:

`routes` Routes paths to protect.

`match` How routes will be matched: exact or partial (as prefix).

`id` Middleware id.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

References

MDN

See Also

Middleware Application

Examples

```
app = Application$new(middleware = list(CORSMiddleware$new()))
app$add_post(path = "/hello", FUN = function(req, res) {
  res$set_body("Hello from RestRserve!")
})
app$add_route("/hello", method = "OPTIONS", FUN = function(req, res) {
 res$set_header("Allow", "POST, OPTIONS")
})
req = Request$new(
  path = "/hello",
  headers = list("Access-Control-Request-Method" = "POST"),
  method = "OPTIONS"
)
app$process_request(req)
```

EncodeDecodeMiddleware

Creates EncodeDecodeMiddleware middleware object

Description

Controls how RestRserve encodes and decodes different content types. **This middleware is passed by default to the Application constructor**.

Super class

`RestRserve::Middleware` -> `EncodeDecodeMiddleware`

Public fields

`ContentHandlers` Class which controls how RestRserve encodes and decodes different content types. See ContentHandlers for documentation. User can add new encoding and decoding methods for new content types using `set_encode` and `set_decode` methods.
In theory user can replace it with his own class (see `RestRserve:::ContentHandlersFactory`). However we believe that in the majority of the cases using ContentHandlers will be enough.

Methods

Public methods:

- `EncodeDecodeMiddleware$new()`
- `EncodeDecodeMiddleware$clone()`

Method `new()`**:** Creates EncodeDecodeMiddleware middleware object.

Usage:

```
EncodeDecodeMiddleware$new(id = "EncodeDecodeMiddleware")
```

Arguments:

`id` Middleware id.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Middleware Application ContentHandlers

`HTTPDate-class` *HTTP Date class*

Description

Conversions between POSIXct to HTTP Date objects.

Arguments

`from`	`numeric`, `POSIXct` or `HTTPDate` object.

References

RFC7231 MDN

Examples

```
# convert POSIXct to HTTP date string
as(0, "HTTPDate") # Thu, 01 Jan 1970 00:00:00 GMT
as(Sys.time(), "HTTPDate")
# parse HTTP date string to POSIXct
dt = "Thu, 01 Jan 1970 00:00:00 GMT"
class(dt) = "HTTPDate"
as(dt, "POSIXct")
```

IDE-hints *request and reponse placeholders for IDE hints*

Description

request and reponse placeholders for IDE hints

Usage

```
.req

.res
```

Format

An object of class `Request` (inherits from `R6`) of length 28.

An object of class `Response` (inherits from `R6`) of length 26.

See Also

Request Response

Examples

```
library(RestRserve)

app = Application$new()

app$add_get("/foo", FUN = function(.req, .res) {
  # since .res is a dummy instance of Response class
  # exported by RestRserve
  # IDE facilitates with autocompletion!
  .res$set_body("bar")
  # in the same time all the modifications happen with local objects
  # so you get right results in the end
})

response = app$process_request(Request$new(path = "/foo"))
response$body
```

Logger *Simple logging utility*

Description

Creates Logger object which can be used for logging with different level of verbosity. Log messages are in JSON format.

Methods

Public methods:

- Logger$new()
- Logger$set_name()
- Logger$set_log_level()
- Logger$set_printer()
- Logger$trace()
- Logger$debug()
- Logger$info()
- Logger$warn()
- Logger$error()
- Logger$fatal()
- Logger$clone()

Method `new()`**:** Creates Logger object.

Usage:

```
Logger$new(
  level = c("info", "fatal", "error", "warn", "debug", "trace", "off", "all"),
  name = "ROOT",
  printer = NULL
)
```

Arguments:

`level` Log level. Allowed values: info, fatal, error, warn, debug, trace, off, all.

`name` Logger name.

`printer` Logger with sink defined by `printer` function. It should have signature function(timestamp, level, logger_name, pid, message). By default when `printer = NULL` logger writes message in JSON format to `stdout`.

Method `set_name()`**:** Sets logger name.

Usage:

```
Logger$set_name(name = "ROOT")
```

Arguments:

`name` Logger name.

Method `set_log_level()`**:** Sets log level.

Usage:

```
Logger$set_log_level(
  level = c("info", "fatal", "error", "warn", "debug", "trace", "off", "all")
)
```

Arguments:

`level` Log level. Allowed values: info, fatal, error, warn, debug, trace, off, all.

Method `set_printer()`**:** Sets function which defines how to print logs.

Usage:

```
Logger$set_printer(FUN = NULL)
```

Arguments:

`FUN` Printer function. Should be a function with 6 formal arguments: timestamp, level, logger_name, pid, message.

Method `trace()`**:** Write trace message.

Usage:

```
Logger$trace(msg, ...)
```

Arguments:

`msg` Log message.

`...` Additionals params.

Method `debug()`**:** Write debug message.

Usage:

```
Logger$debug(msg, ...)
```

Arguments:

`msg` Log message.

`...` Additionals params.

Method `info()`**:** Write information message.

Usage:

```
Logger$info(msg, ...)
```

Arguments:

`msg` Log message.

`...` Additionals params.

Method `warn()`**:** Write warning message.

Usage:

```
Logger$warn(msg, ...)
```

Arguments:

`msg` Log message.

`...` Additionals params.

Method `error()`**:** Write error message.

Usage:

```
Logger$error(msg, ...)
```

Arguments:

`msg` Log message.

`...` Additionals params.

Method `fatal()`**:** Write fatal error message.

Usage:

```
Logger$fatal(msg, ...)
```

Arguments:

`msg` Log message.

`...` Additionals params.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

lgr::Logger

Examples

```
# init logger
logger = Logger$new("info")
# write info message
logger$info("hello world")
# write extended log entry
logger$info("", context = list(message = "hello world", code = 0L))
```

`Middleware` *Creates middleware object*

Description

Creates Middleware object.

Middleware is a very useful concept which allows to perform preprocessing of requests and post-processing of responses. Middleware has an access to both `request` and `response` objects and can modify them. This way each request can be checked/modified before passing handler and response can be post processed (for example this way we developer can set up custom error messages).

Public fields

`process_request` Function which takes 2 arguments - `request` and `response` objects (class Request and Response correspondingly) and modify `request` and `response` or throw exception using HTTPError helper.
Function is called before request is routed to handler.
Usually `process_request` is used to perform logging, check authorization, etc.

`process_response` Function which takes 2 arguments - `request` and `response` objects (class Request and Response correspondingly) and modify `request` and `response` or throw exception using HTTPError helper.
Function is called after request is processed by handler. Usually `process_response` is used to perform logging, custom error handling, etc.

`id` Middleware id.

Methods

Public methods:

- `Middleware$new()`
- `Middleware$clone()`

Method `new()`**:** Creates middleware object

Usage:

```
Middleware$new(
  process_request = function(request, response) TRUE,
  process_response = function(request, response) TRUE,
  id = "Middleware"
)
```

Arguments:

`process_request` Modify `request` or `response` objects or throw exception using [HTTPError] helper. This function evaluate before router handler called.

`process_response` Modify `request` or `response` objects or throw exception using [HTTPError] helper. This function evaluate after router handler called.

`id` Middleware id.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

See Also

Request Response Application

raise *Interrupts request processing*

Description

Interrupts request processing and signals RestRserve to return HTTPError

Usage

```
raise(x)
```

Arguments

x	instance of Response. Can be created using HTTPError. see examples.

Value

None - stops execution of the current expression and executes an error action.

See Also

HTTPError Application

Examples

```
# catch exception
res = try(raise(HTTPError$bad_request()), silent = TRUE)
cond = attr(res, "condition")
# response is a valid Response instace
identical(cond$response$body$error, "400 Bad Request")
```

Request *Creates Request object*

Description

Called internally for handling incoming requests from Rserve side. Also useful for testing.

Public fields

`path` Request path.

`method` Request HTTP method.

`headers` Request headers.

`cookies` Request cookies.

`context` Environment to store any data. Can be used in middlewares.

`content_type` Request body content type.

`body` Request body.

`parameters_query` Request query parameters.

`parameters_body` Request body parameters.

`parameters_path` List of parameters extracted from templated path after routing. For example if we have some handler listening at /job/{job_id} and we are receiving request at /job/1 then `parameters_path` will be `list(job_id = "1")`.
It is important to understand that `parameters_path` will be available (not empty) only after request will reach handler.
This effectively means that `parameters_path` can be used inside handler and response middleware (but not request middleware!).

`files` Structure which contains positions and lengths of files for the multipart body.

`decode` Function to decode body for the specific content type.

Active bindings

`id` Automatically generated UUID for each request. Read only.

`date` Request `Date` header converted to `POSIXct`.

`accept` Splitted `Accept` request header.

`accept_json` Request accepts JSON response.

`accept_xml` Request accepts XML response.

Methods

Public methods:

- `Request$new()`
- `Request$set_id()`
- `Request$reset()`
- `Request$get_header()`
- `Request$get_param_query()`
- `Request$get_param_body()`
- `Request$get_param_path()`
- `Request$get_file()`
- `Request$print()`
- `Request$clone()`

Method `new()`**:** Creates Request object

Usage:

```
Request$new(
  path = "/",
 method = c("GET", "HEAD", "POST", "PUT", "DELETE", "CONNECT", "OPTIONS", "TRACE",
    "PATCH"),
  parameters_query = list(),
  parameters_body = list(),
  headers = list(),
  body = NULL,
  cookies = list(),
  content_type = NULL,
  decode = NULL,
  ...
)
```

Arguments:

`path` Character with requested path. Always starts with `/`.

`method` Request HTTP method.

`parameters_query` A named list with URL decoded query parameters.

`parameters_body` A named list with URL decoded body parameters. This field is helpful when request is a urlencoded form or a multipart form.

`headers` Request HTTP headers represented as named list.

`body` Request body. Can be anything and in conjunction with `content_type` defines how HTTP body will be represented.

`cookies` Cookies represented as named list. **Note** that cookies should be provided explicitly - they won't be derived from `headers`.

`content_type` HTTP content type. **Note** that `content_type` should be provided explicitly - it won't be derived from `headers`.

`decode` Function to decode body for the specific content type.

`...` Not used at this moment.

Method `set_id()`**:** Set request id.

Usage:

```
Request$set_id(id = uuid::UUIDgenerate(TRUE))
```

Arguments:

`id` Request id.

Method `reset()`**:** Resets request object. This is not useful for end user, but useful for RestRserve internals - resetting R6 class is much faster then initialize it.

Usage:

```
Request$reset()
```

Method `get_header()`**:** Get HTTP response header value. If requested header is empty returns `default`.

Usage:

```
Request$get_header(name, default = NULL)
```

Arguments:

`name` Header field name.

`default` Default value if header does not exists.

Returns: Header field values (character string).

Method `get_param_query()`**:** Get request query parameter by name.

Usage:

```
Request$get_param_query(name)
```

Arguments:

`name` Query parameter name.

Returns: Query parameter value (character string).

Method `get_param_body()`**:** Get request body parameter by name.

Usage:

```
Request$get_param_body(name)
```

Arguments:

`name` Body field name.

Returns: Body field value.

Method `get_param_path()`**:** Get templated path parameter by name.

Usage:

```
Request$get_param_path(name)
```

Arguments:

`name` Path parameter name.

Returns: Path parameter value.

Method `get_file()`**:** Extract specific file from multipart body.

Usage:

```
Request$get_file(name)
```

Arguments:

`name` Body file name.

Returns: Raw vector with `filname` and `content-type` attributes.

Method `print()`**:** Print method.

Usage:

```
Request$print()
```

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.

See Also

Response Application

Examples

```
# init simply request
rq = Request$new(
  path = "/",
  parameters_query = list(
    "param1" = "value1",
    "param2" = "value2"
  ),
  headers = list(
    "Content-encoding" = "identity",
    "Custom-field" = "value"
  ),
  cookies = list(
    "sessionId" = "1"
  )
)
# get request UUID
rq$id
# get content accept
rq$accept
# get request content type
rq$content_type
# get header by name (lower case)
rq$get_header("custom-field")
# get query param by name
rq$get_param_query("param1")
# print request
rq
```

Response *Creates Response object*

Description

Creates response object.

Public fields

`body` Response body.
If it is a named character with a name `file` or `tmpfile` then the value is considered as a path to a file and content oh this file is served as body. The latter will be deleted once served.

`content_type` Response body content (media) type. Will be translated to `Content-type` header.

`headers` Response headers.

`status_code` Response HTTP status code.

`cookies` Response cookies. Will be translated to `Set-Cookie` headers.

`context` Environment to store any data. Can be used in middlewares.

`encode` Function to encode body for specific content.

Active bindings

`status` Paste together status code and description.

Methods

Public methods:

- `Response$new()`
- `Response$reset()`
- `Response$set_content_type()`
- `Response$set_status_code()`
- `Response$has_header()`
- `Response$get_header()`
- `Response$set_header()`
- `Response$delete_header()`
- `Response$append_header()`
- `Response$set_date()`
- `Response$unset_date()`
- `Response$set_cookie()`
- `Response$unset_cookie()`
- `Response$set_body()`
- `Response$set_response()`
- `Response$print()`
- `Response$clone()`

Method `new()`**:** Creates Response object

Usage:

```
Response$new(
  body = NULL,
  content_type = "text/plain",
  headers = list(Server = getOption("RestRserve.headers.server")),
  status_code = 200L,
  encode = NULL,
  ...
)
```

Arguments:

`body` Response body.

`content_type` Response body content (media) type.

`headers` Response headers.

`status_code` Response status code.

`encode` Function to encode body for specific content.

`...` Not used at this moment.

Method `reset()`**:** Resets response object. This is not useful for end user, but useful for RestRserve internals - resetting R6 class is much faster then initialize it.

Usage:

```
Response$reset()
```

Method `set_content_type()`**:** Set content type for response body.

Usage:

```
Response$set_content_type(content_type = "text/plain")
```

Arguments:

`content_type` Response body content (media) type.

Method `set_status_code()`**:** Set HTTP status code for response. See docs on MDN.

Usage:

```
Response$set_status_code(code)
```

Arguments:

`code` Status code as integer number.

Method `has_header()`**:** Determine whether or not the response header exists.

Usage:

```
Response$has_header(name)
```

Arguments:

`name` Header field name.

Returns: Logical value.

Method `get_header()`**:** Get HTTP response header value. If requested header is empty returns `default`.

Usage:

```
Response$get_header(name, default = NULL)
```

Arguments:

`name` Header field name.

`default` Default value if header does not exists.

Returns: Header field values (character string).

Method `set_header()`**:** Set HTTP response header. `Content-type` and `Content-length` headers not allowed (use `content_type` field instead).

Usage:

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

Arguments:

`name` Header field name.

`value` Header field value.

Method `delete_header()`**:** Unset HTTP response header.

Usage:

```
Response$delete_header(name)
```

Arguments:

`name` Header field name.

Returns: Logical value.

Method `append_header()`**:** Append HTTP response header. If header exists , separator will be used. Don't use this method to set cookie (use `set_cookie` method instead).

Usage:

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

Arguments:

`name` Header field name.

`value` Header field value.

Method `set_date()`**:** Set `Date` HTTP header. See docs on MDN.

Usage:

```
Response$set_date(dtm = Sys.time())
```

Arguments:

`dtm` POSIXct value.

Method `unset_date()`**:** Unset `Date` HTTP header.

Usage:

```
Response$unset_date()
```

Returns: Logical value.

Method `set_cookie()`**:** Set cookie. See docs on MDN.

Usage:

```
Response$set_cookie(
  name,
  value,
  expires = NULL,
  max_age = NULL,
  domain = NULL,
  path = NULL,
  secure = NULL,
  http_only = NULL
)
```

Arguments:

`name` Cookie name.

`value` Cookie value.

`expires` Cookie expires date and time (POSIXct).

`max_age` Max cookie age (integer).

`domain` Cookie domain.

`path` Cookie path.

`secure` Cookie secure flag.

`http_only` Cookie HTTP only flag.

Method `unset_cookie()`**:** Unset cookie with given name.

Usage:

```
Response$unset_cookie(name)
```

Arguments:

`name` Cookie name.

Returns: Logical value.

Method `set_body()`**:** Set response body.

Usage:

```
Response$set_body(body)
```

Arguments:

`body` Response body.

Method `set_response()`**:** Set response fields.

Usage:

```
Response$set_response(
  status_code,
  body = NULL,
  content_type = self$content_type
)
```

Arguments:

`status_code` Response HTTP status code.

`body` Response body.

`content_type` content_type Response body content (media) type.

Method `print()`**:** Print method.

Usage:

```
Response$print()
```

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.

See Also

Request Application

Examples

```
# init response
rs = Response$new()
# set body media type
rs$set_content_type("text/plain")
# set body content
rs$set_body("OK")
# set response status code
rs$set_status_code(200L)
# print response
rs

# init response
rs = Response$new()
# static file path
file_path = system.file("DESCRIPTION", package = "RestRserve")
# get last file modification timestamp
file_mtime = file.mtime(file_path)
# set body
rs$set_body(c("file" = file_path))
# set content type
rs$set_content_type("text/plain")
# set current timestamp
rs$set_date()
# set 'last-modified' header
rs$set_header("Last-Modified", as(file_mtime, "HTTPDate"))
# print response
rs
```

---

to_json *Simple JSON encoder*

Description

Encode R objects as JSON. Wrapper around `jsonlite::toJSON` with default parameters set to following values: dataframe = 'columns', auto_unbox = unbox, null = 'null', na = 'null'.

Usage

```
to_json(x, unbox = TRUE)
```

Arguments

x	the object to be encoded
unbox	TRUE by default. Whether to unbox (simplify) arrays consists of a single element

Value

JSON string

Examples

```
to_json(NULL)
to_json(list(name = "value"))
```

Index