Package 'plumber'

September 14, 2020

Encoding UTF-8

Type Package

Title An API Generator for R

Version 1.0.0

Roxygen list(markdown = TRUE)

License MIT + file LICENSE

BugReports https://github.com/rstudio/plumber/issues

URL https://www.rplumber.io, https://github.com/rstudio/plumber

Description Gives the ability to automatically generate and serve an HTTP API from R functions using the annotations in the R documentation around your functions.

Depends R (>= 3.0.0)

Imports R6 (>= 2.0.0),
stringi (>= 0.3.0),
jsonlite (>= 0.9.16),
webutils (>= 1.1),
httpuv (>= 1.5.0),
crayon,
promises (>= 1.1.0),
sodium,
swagger (>= 3.33.0),
magrittr,
mime,
lifecycle

LazyData TRUE

ByteCompile TRUE

Suggests testthat (>= 0.11.0),
rmarkdown,
base64enc,
htmlwidgets,
visNetwork,
later,
readr,
yaml,
feather,
future

RoxygenNote 7.1.1

Collate 'async.R'
'content-types.R'
'default-handlers.R'
'hookable.R'
'shared-secret-filter.R'
'parser-cookie.R'
'parse-body.R'
'parse-query.R'
'plumber.R'
'deprecated-R6.R'
'deprecated.R'
'digital-ocean.R'
'find-port.R'
'globals.R'
'includes.R'
'json.R'
'new-rstudio-project.R'
'openapi-spec.R'
'openapi-types.R'
'paths.R'
'plumb-block.R'
'plumb-globals.R'
'plumb.R'
'plumber-options.R'
'plumber-response.R'
'plumber-static.R'
'plumber-step.R'
'pr.R'
'pr_set.R'
'serializer.R'
'session-cookie.R'
'ui.R'
'utf8.R'
'utils-pipe.R'
'utils.R'
'validate_api_spec.R'
'zzz.R'

RdMacros lifecycle

R topics documented:

`as_attachment` *Return an attachment response*

Description

This will set the appropriate fields in the `Content-Disposition` header value. To make sure the attachment is used, be sure your serializer eventually calls `serializer_headers`

Usage

```
as_attachment(value, filename = NULL)
```

Arguments

`value`	Response value to be saved
`filename`	File name to use when saving the attachment. If no `filename` is provided, the `value` will be treated as a regular attachment

Value

Object with class `"plumber_attachment"`

Examples

```
# plumber.R

#' @get /data
#' @serializer csv
function() {
  # will cause the file to be saved as `iris.csv`, not `data` or `data.csv`
  as_attachment(iris, "iris.csv")
}
```

`endpoint_serializer` *Endpoint Serializer with Hooks*

Description

This method allows serializers to return `preexec`, `postexec`, and `aroundexec` (**Experimental**) hooks in addition to a serializer. This is useful for graphics device serializers which need a `preexec` and `postexec` hook to capture the graphics output.

Usage

```
endpoint_serializer(
  serializer,
  preexec_hook = NULL,
  postexec_hook = NULL,
  aroundexec_hook = NULL
)
```

Arguments

`serializer`	Serializer method to be used. This method should already have its initialization arguments applied.
`preexec_hook`	Function to be run directly before a PlumberEndpoint calls its route method.
`postexec_hook`	Function to be run directly after a PlumberEndpoint calls its route method.
`aroundexec_hook`	Function to be run around a PlumberEndpoint call. Must handle a `.next` argument to continue execution. **Experimental**

Details

`preexec` and `postexec` hooks happened directly before and after a route is executed. These hooks are specific to a single PlumberEndpoint's route calculation.

Examples

```
# The definition of `serializer_device` returns
# * a `serializer_content_type` serializer
# * `aroundexec` hook
print(serializer_device)
```

forward *Forward Request to The Next Handler*

Description

This function is used when a filter is done processing a request and wishes to pass control off to the next handler in the chain. If this is not called by a filter, the assumption is that the filter fully handled the request itself and no other filters or endpoints should be evaluated for this request.

Usage

```
forward()
```

get_character_set *Request character set*

Description

Request character set

Usage

```
get_character_set(content_type = NULL)
```

Arguments

content_type	Request Content-Type header

Value

Default to UTF-8. Otherwise return charset defined in request header.

include_file *Send File Contents as Response*

Description

Returns the file at the given path as the response.

Usage

```
include_file(file, res, content_type = getContentType(tools::file_ext(file)))

include_html(file, res)

include_md(file, res, format = NULL)

include_rmd(file, res, format = NULL)
```

Arguments

file	The path to the file to return
res	The response object into which we'll write
content_type	If provided, the given value will be sent as the Content-Type header in the response. Defaults to the contentType of the file extension. To disable the Content-Type header, set content_type = NULL.
format	Passed as the output_format to rmarkdown::render

Details

include_html will merely return the file with the proper content_type for HTML. include_md and include_rmd will process the given markdown file through rmarkdown::render and return the resultant HTML as a response.

---

is_plumber *Determine if Plumber object*

Description

Determine if Plumber object

Usage

```
is_plumber(pr)
```

Arguments

pr	Hopefully a Plumber object

Value

Logical value if pr inherits from Plumber

Examples

```
is_plumber(Plumber$new()) # TRUE
is_plumber(list()) # FALSE
```

options_plumber *Plumber options*

Description

There are a number of global options that affect Plumber's behavior. These can be set globally with `options()` or with `options_plumber()`. Options set using `options_plumber()` should not include the `plumber.` prefix.

Usage

```
options_plumber(
  port = getOption("plumber.port"),
  docs = getOption("plumber.docs"),
  docs.callback = getOption("plumber.docs.callback"),
  apiURL = getOption("plumber.apiURL"),
  apiScheme = getOption("plumber.apiScheme"),
  apiHost = getOption("plumber.apiHost"),
  apiPort = getOption("plumber.apiPort"),
  apiPath = getOption("plumber.apiPath"),
  maxRequestSize = getOption("plumber.maxRequestSize"),
  sharedSecret = getOption("plumber.sharedSecret")
)
```

Arguments

`port`, `docs`, `docs.callback`, `apiScheme`, `apiHost`, `apiPort`, `apiPath`, `apiURL`, `maxRequestSize`, `sharedSecret` See details

Details

`plumber.port` Port Plumber will attempt to use to start http server. If the port is already in use, server will not be able to start. Defaults to `NULL`

`plumber.docs` Name of the visual documentation interface to use. Defaults to `TRUE`, which will use `"swagger"`

`plumber.docs.callback` A function. Called with a single parameter corresponding to the visual documentation url after Plumber server is ready. This can be used by RStudio to open the docs when then API is ran from the editor. Defaults to option `NULL`

`plumber.apiURL` Server urls for OpenAPI Specification respecting pattern scheme://host:port/path. Other api* options will be ignored when set.

`plumber.apiScheme` Scheme used to build OpenAPI url and server url for OpenAPI Specification. Defaults to `http`, or an empty string when used outside a running router

`plumber.apiHost` Host used to build docs url and server url for OpenAPI Specification. Defaults to `host` defined by `run` method, or an empty string when used outside a running router

`plumber.apiPort` Port used to build OpenAPI url and server url for OpenAPI Specification. Defaults to `port` defined by `run` method, or an empty string when used outside a running router

`plumber.apiPath` Path used to build OpenAPI url and server url for OpenAPI Specification. Defaults to an empty string

`plumber.maxRequestSize` Maximum length in bytes of request body. Body larger than maximum are rejected with http error 413. `0` means unlimited size. Defaults to `0`

`plumber.sharedSecret` Shared secret used to filter incoming request. When `NULL`, secret is not validated. Otherwise, Plumber compares secret with http header `PLUMBER_SHARED_SECRET`. Failure to match results in http error 400. Defaults to `NULL`

Value

The complete, prior set of `options()` values. If a particular parameter is not supplied, it will return the current value. If no parameters are supplied, all returned values will be the current `options()` values.

`parser_form` *Plumber Parsers*

Description

Parsers are used in Plumber to transform the raw body content received by a request to the API. Extra parameters may be provided to parser functions when adding the parser to plumber. This will allow for non-default behavior.

Usage

```
parser_form()

parser_json(...)

parser_text(parse_fn = identity)

parser_yaml(...)

parser_csv(...)

parser_tsv(...)

parser_read_file(read_fn = readLines)

parser_rds(...)

parser_feather(...)

parser_octet()

parser_multi()

parser_none()
```

Arguments

`...`	parameters supplied to the appropriate internal function
`parse_fn`	function to further decode a text string into an object
`read_fn`	function used to read a the content of a file. Ex: `readRDS()`

Details

Parsers are optional. When unspecified, only the `parser_json()`, `parser_octet()`, `parser_form()` and `parser_text()` are available. You can use @parser parser tag to activate parsers per endpoint. Multiple parsers can be activated for the same endpoint using multiple @parser parser tags.

User should be aware that `rds` parsing should only be done from a trusted source. Do not accept `rds` files blindly.

See `registered_parsers()` for a list of registered parsers.

Functions

- `parser_form`: Form query string parser
- `parser_json`: JSON parser. See `jsonlite::parse_json()` for more details. (Defaults to using `simplifyVectors = TRUE`)
- `parser_text`: Helper parser to parse plain text
- `parser_yaml`: YAML parser. See `yaml::yaml.load()` for more details.
- `parser_csv`: CSV parser. See `readr::read_csv()` for more details.
- `parser_tsv`: TSV parser. See `readr::read_tsv()` for more details.
- `parser_read_file`: Helper parser that writes the binary body to a file and reads it back again using `read_fn`. This parser should be used when reading from a file is required.
- `parser_rds`: RDS parser. See `readRDS()` for more details.
- `parser_feather`: feather parser. See `feather::read_feather()` for more details.
- `parser_octet`: Octet stream parser. Returns the raw content.
- `parser_multi`: Multi part parser. This parser will then parse each individual body with its respective parser. When this parser is used, `req$body` will contain the updated output from `webutils::parse_multipart()` by adding the `parsed` output to each part. Each part may contain detailed information, such as `name` (required), `content_type`, `content_disposition`, `filename`, (raw, original) `value`, and `parsed` (parsed `value`). When performing Plumber route argument matching, each multipart part will match its `name` to the `parsed` content.
- `parser_none`: No parser. Will not process the postBody.

Examples

```
## Not run:
# Overwrite `text/json` parsing behavior to not allow JSON vectors to be simplified
#* @parser json simplifyVector = FALSE
# Activate `rds` parser in a multipart request
#* @parser multi
#* @parser rds
pr <- Plumber$new()
pr$handle("GET", "/upload", function(rds) {rds}, parsers = c("multi", "rds"))

## End(Not run)
```

plumb *Process a Plumber API*

Description

Process a Plumber API

Usage

```
plumb(file = NULL, dir = ".")
```

Arguments

file	The file to parse as the plumber router definition.
dir	The directory containing the plumber.R file to parse as the plumber router definition. Alternatively, if an entrypoint.R file is found, it will take precedence and be responsible for returning a runnable router.

Details

API routers are the core request handler in plumber. A router is responsible for taking an incoming request, submitting it through the appropriate filters and eventually to a corresponding endpoint, if one is found.

See https://www.rplumber.io/articles/programmatic-usage.html for additional details on the methods available on this object.

Plumber *Package Plumber Router*

Description

Package Plumber Router

Package Plumber Router

Details

Routers are the core request handler in plumber. A router is responsible for taking an incoming request, submitting it through the appropriate filters and eventually to a corresponding endpoint, if one is found.

See https://www.rplumber.io/articles/programmatic-usage.html for additional details on the methods available on this object.

Super class

plumber::Hookable -> Plumber

Active bindings

`endpoints` plumber router endpoints read-only

`filters` plumber router filters read-only

`mounts` plumber router mounts read-only

`environment` plumber router environment read-only

`routes` plumber router routes read-only

Methods

Public methods:

- `Plumber$new()`
- `Plumber$run()`
- `Plumber$mount()`
- `Plumber$unmount()`
- `Plumber$registerHook()`
- `Plumber$handle()`
- `Plumber$removeHandle()`
- `Plumber$print()`
- `Plumber$serve()`
- `Plumber$route()`
- `Plumber$call()`
- `Plumber$onHeaders()`
- `Plumber$onWSOpen()`
- `Plumber$setSerializer()`
- `Plumber$setParsers()`
- `Plumber$set404Handler()`
- `Plumber$setErrorHandler()`
- `Plumber$setDocs()`
- `Plumber$setDocsCallback()`
- `Plumber$setDebug()`
- `Plumber$getDebug()`
- `Plumber$filter()`
- `Plumber$setApiSpec()`
- `Plumber$getApiSpec()`
- `Plumber$addEndpoint()`
- `Plumber$addAssets()`
- `Plumber$addFilter()`
- `Plumber$addGlobalProcessor()`
- `Plumber$openAPIFile()`
- `Plumber$swaggerFile()`
- `Plumber$clone()`

Method `new()`**:** Create a new `Plumber` router

See also `plumb()`, `pr()`

Usage:

```
Plumber$new(file = NULL, filters = defaultPlumberFilters, envir)
```

Arguments:

`file` path to file to plumb

`filters` a list of Plumber filters

`envir` an environment to be used as the enclosure for the routers execution

Returns: A new `Plumber` router

Method `run()`**:** Start a server using `plumber` object.

See also: `pr_run()`

Usage:

```
Plumber$run(
  host = "127.0.0.1",
  port = getOption("plumber.port", NULL),
  swagger = deprecated(),
  debug = deprecated(),
  swaggerCallback = deprecated()
)
```

Arguments:

`host` a string that is a valid IPv4 or IPv6 address that is owned by this server, which the application will listen on. "0.0.0.0" represents all IPv4 addresses and "::/0" represents all IPv6 addresses.

`port` a number or integer that indicates the server port that should be listened on. Note that on most Unix-like systems including Linux and Mac OS X, port numbers smaller than 1025 require root privileges.
This value does not need to be explicitly assigned. To explicitly set it, see `options_plumber()`.

`swagger` Deprecated. See $setDocs(docs) or $setApiSpec()

`debug` Deprecated. See $setDebug()

`swaggerCallback` Deprecated. See $setDocsCallback()

Method `mount()`**:** Mount a Plumber router

Plumber routers can be "nested" by mounting one into another using the `mount()` method. This allows you to compartmentalize your API by paths which is a great technique for decomposing large APIs into smaller files.

See also: `pr_mount()`

Usage:

```
Plumber$mount(path, router)
```

Arguments:

`path` a character string. Where to mount router.

`router` a Plumber router. Router to be mounted.

Examples:

```
\dontrun{
root <- pr()

users <- Plumber$new("users.R")
root$mount("/users", users)

products <- Plumber$new("products.R")
root$mount("/products", products)
}
```

Method `unmount()`: Unmount a Plumber router

Usage:

```
Plumber$unmount(path)
```

Arguments:

`path` a character string. Where to unmount router.

Method `registerHook()`: Register a hook

Plumber routers support the notion of "hooks" that can be registered to execute some code at a particular point in the lifecycle of a request. Plumber routers currently support four hooks:

1. `preroute(data,req,res)`
2. `postroute(data,req,res,value)`
3. `preserialize(data,req,res,value)`
4. `postserialize(data,req,res,value)`

In all of the above you have access to a disposable environment in the `data` parameter that is created as a temporary data store for each request. Hooks can store temporary data in these hooks that can be reused by other hooks processing this same request.

One feature when defining hooks in Plumber routers is the ability to modify the returned value. The convention for such hooks is: any function that accepts a parameter named `value` is expected to return the new value. This could be an unmodified version of the value that was passed in, or it could be a mutated value. But in either case, if your hook accepts a parameter named `value`, whatever your hook returns will be used as the new value for the response.

You can add hooks using the `registerHook` method, or you can add multiple hooks at once using the `registerHooks` method which takes a name list in which the names are the names of the hooks, and the values are the handlers themselves.

See also: `pr_hook()`, `pr_hooks()`

Usage:

```
Plumber$registerHook(
  stage = c("preroute", "postroute", "preserialize", "postserialize", "exit"),
  handler
)
```

Arguments:

`stage` a character string. Point in the lifecycle of a request.

`handler` a hook function.

Examples:

```
\dontrun{
pr <- pr()
pr$registerHook("preroute", function(req){
  cat("Routing a request for", req$PATH_INFO, "...\n")
})
pr$registerHooks(list(
  preserialize=function(req, value){
    print("About to serialize this value:")
    print(value)

    # Must return the value since we took one in. Here we're not choosing
    # to mutate it, but we could.
    value
  },
```

```
  postserialize=function(res){
    print("We serialized the value as:")
    print(res$body)
  }
))

pr$handle("GET", "/", function(){ 123 })
}
```

Method `handle()`**:** Define endpoints

The "handler" functions that you define in these handle calls are identical to the code you would have defined in your plumber.R file if you were using annotations to define your API. The handle() method takes additional arguments that allow you to control nuanced behavior of the endpoint like which filter it might preempt or which serializer it should use.

See also: `pr_handle()`, `pr_get()`, `pr_post()`, `pr_put()`, `pr_delete()`

Usage:

```
Plumber$handle(
  methods,
  path,
  handler,
  preempt,
  serializer,
  parsers,
  endpoint,
  ...
)
```

Arguments:

`methods` a character string. http method.

`path` a character string. Api endpoints

`handler` a handler function.

`preempt` a preempt function.

`serializer` a serializer function.

`parsers` a named list of parsers.

`endpoint` a `PlumberEndpoint` object.

`...` additional arguments for `PlumberEndpoint` creation

Examples:

```
\dontrun{
pr <- pr()
pr$handle("GET", "/", function(){
  "<html><h1>Programmatic Plumber!</h1></html>"
}, serializer=plumber::serializer_html())
}
```

Method `removeHandle()`**:** Remove endpoints

Usage:

```
Plumber$removeHandle(methods, path, preempt = NULL)
```

Arguments:

`methods` a character string. http method.

`path` a character string. Api endpoints

`preempt` a preempt function.

Method `print()`**:** Print representation of plumber router.

Usage:

```
Plumber$print(prefix = "", topLevel = TRUE, ...)
```

Arguments:

`prefix` a character string. Prefix to append to representation.

`topLevel` a logical value. When method executed on top level router, set to `TRUE`.

`...` additional arguments for recursive calls

Returns: A terminal friendly representation of a plumber router.

Method `serve()`**:** Serve a request

Usage:

```
Plumber$serve(req, res)
```

Arguments:

`req` request object

`res` response object

Method `route()`**:** Route a request

Usage:

```
Plumber$route(req, res)
```

Arguments:

`req` request object

`res` response object

Method `call()`**:** **httpuv** interface call function. (Required for **httpuv**)

Usage:

```
Plumber$call(req)
```

Arguments:

`req` request object

Method `onHeaders()`**:** httpuv interface onHeaders function. (Required for **httpuv**)

Usage:

```
Plumber$onHeaders(req)
```

Arguments:

`req` request object

Method `onWSOpen()`**:** httpuv interface onWSOpen function. (Required for **httpuv**)

Usage:

```
Plumber$onWSOpen(ws)
```

Arguments:

`ws` WebSocket object

Method `setSerializer()`**:** Sets the default serializer of the router.

See also: `pr_set_serializer()`

Usage:

```
Plumber$setSerializer(serializer)
```

Arguments:

`serializer` a serializer function

Examples:

```
\dontrun{
pr <- pr()
pr$setSerializer(serializer_unboxed_json())
}
```

Method `setParsers()`**:** Sets the default parsers of the router. Initialized to `c("json","form","text","octet","mu`

Usage:

```
Plumber$setParsers(parsers)
```

Arguments:

`parsers` Can be one of:

- A `NULL` value
- A character vector of parser names
- A named `list()` whose keys are parser names names and values are arguments to be applied with `do.call()`
- A `TRUE` value, which will default to combining all parsers. This is great for seeing what is possible, but not great for security purposes

If the parser name `"all"` is found in any character value or list name, all remaining parsers will be added. When using a list, parser information already defined will maintain their existing argument values. All remaining parsers will use their default arguments.
Example:

```
# provide a character string
parsers = "json"

# provide a named list with no arguments
parsers = list(json = list())

# provide a named list with arguments; include `rds`
parsers = list(json = list(simplifyVector = FALSE), rds = list())

# default plumber parsers
parsers = c("json", "form", "text", "octet", "multi")
```

Method `set404Handler()`**:** Sets the handler that gets called if an incoming request can't be served by any filter, endpoint, or sub-router.

See also: `pr_set_404()`

Usage:

```
Plumber$set404Handler(fun)
```

Arguments:

`fun` a handler function.

Examples:

```
\dontrun{
pr <- pr()
pr$set404Handler(function(req, res) {cat(req$PATH_INFO)})
}
```

Method `setErrorHandler()`**:** Sets the error handler which gets invoked if any filter or endpoint generates an error.

See also: `pr_set_404()`

Usage:

```
Plumber$setErrorHandler(fun)
```

Arguments:

`fun` a handler function.

Examples:

```
\dontrun{
pr <- pr()
pr$setErrorHandler(function(req, res, err) {
  message("Found error: ")
  str(err)
})
}
```

Method `setDocs()`**:** Set visual documentation to use for API

See also: `pr_set_docs()`, `register_docs()`, `registered_docs()`

Usage:

```
Plumber$setDocs(docs = getOption("plumber.docs", TRUE), ...)
```

Arguments:

`docs` a character value or a logical value. See `pr_set_docs()` for examples. If using `options_plumber()`, the value must be set before initializing your Plumber router.

`...` Other params to be passed to `docs` functions.

Method `setDocsCallback()`**:** Set a callback to notify where the API's visual documentation is located.

When set, it will be called with a character string corresponding to the API docs url. This allows RStudio to open `swagger` docs when a Plumber router `pr_run()` method is executed.

If using `options_plumber()`, the value must be set before initializing your Plumber router.

See also: `pr_set_docs_callback()`

Usage:

```
Plumber$setDocsCallback(callback = getOption("plumber.docs.callback", NULL))
```

Arguments:

`callback` a callback function for taking action on the docs url. (Also accepts `NULL` values to disable the `callback`.)

Method `setDebug()`**:** Set debug value to include error messages

See also: $getDebug() and `pr_set_debug()`

Usage:

```
Plumber$setDebug(debug = interactive())
```

Arguments:

`debug` `TRUE` provides more insight into your API errors.

Method `getDebug()`**:** Retrieve the `debug` value.

See also: $getDebug() and `pr_set_debug()`

Usage:

```
Plumber$getDebug()
```

Method `filter()`**:** Add a filter to plumber router

See also: `pr_filter()`

Usage:

```
Plumber$filter(name, expr, serializer)
```

Arguments:

`name` a character string. Name of filter

`expr` an expr that resolve to a filter function or a filter function

`serializer` a serializer function

Method `setApiSpec()`**:** Add a function to customize what is returned in $getApiSpec(). Note, the returned value will be sent through `serializer_unboxed_json()` which will turn all length 1 vectors into atomic values. To force a vector to serialize to an array of size 1, be sure to call `as.list()` on your value. `list()` objects are always serialized to an array value.

See also: `pr_set_api_spec()`

Usage:

```
Plumber$setApiSpec(api = NULL)
```

Arguments:

`api` This can be

- an OpenAPI Specification formatted list object
- a function that accepts the OpenAPI Specification autogenerated by `plumber` and returns a OpenAPI Specification formatted list object.

The value returned will not be validated for OAS compatibility.

Method `getApiSpec()`**:** Retrieve openAPI file

Usage:

```
Plumber$getApiSpec()
```

Method `addEndpoint()`**:** addEndpoint has been deprecated in v0.4.0 and will be removed in a coming release. Please use `handle()` instead.

Usage:

```
Plumber$addEndpoint(
  verbs,
  path,
  expr,
  serializer,
  processors,
  preempt = NULL,
  params = NULL,
  comments
)
```

Arguments:

`verbs` verbs

`path` path

`expr` expr

`serializer` serializer

`processors` processors

preempt preempt

params params

comments comments

Method `addAssets()`: addAssets has been deprecated in v0.4.0 and will be removed in a coming release. Please use `mount` and `PlumberStatic$new()` instead.

Usage:

```
Plumber$addAssets(dir, path = "/public", options = list())
```

Arguments:

dir dir

path path

options options

Method `addFilter()`: $addFilter() has been deprecated in v0.4.0 and will be removed in a coming release. Please use $filter() instead.

Usage:

```
Plumber$addFilter(name, expr, serializer, processors)
```

Arguments:

name name

expr expr

serializer serializer

processors processors

Method `addGlobalProcessor()`: $addGlobalProcessor() has been deprecated in v0.4.0 and will be removed in a coming release. Please use $registerHook(s) instead.

Usage:

```
Plumber$addGlobalProcessor(proc)
```

Arguments:

proc proc

Method `openAPIFile()`: Deprecated. Retrieve openAPI file

Usage:

```
Plumber$openAPIFile()
```

Method `swaggerFile()`: Deprecated. Retrieve openAPI file

Usage:

```
Plumber$swaggerFile()
```

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

Usage:

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

Arguments:

deep Whether to make a deep clone.

See Also

`pr()`, `pr_run()`, `pr_get()`, `pr_post()`, `pr_mount()`, `pr_hook()`, `pr_hooks()`, `pr_cookie()`, `pr_filter()`, `pr_set_api_spec()`, `pr_set_docs()`, `pr_set_serializer()`, `pr_set_parsers()`, `pr_set_404()`, `pr_set_error()`, `pr_set_debug()`, `pr_set_docs_callback()`

Examples

```
## ------------------------------------------------
## Method `Plumber$mount`
## ------------------------------------------------

## Not run:
root <- pr()

users <- Plumber$new("users.R")
root$mount("/users", users)

products <- Plumber$new("products.R")
root$mount("/products", products)

## End(Not run)

## ------------------------------------------------
## Method `Plumber$registerHook`
## ------------------------------------------------

## Not run:
pr <- pr()
pr$registerHook("preroute", function(req){
  cat("Routing a request for", req$PATH_INFO, "...\n")
})
pr$registerHooks(list(
  preserialize=function(req, value){
    print("About to serialize this value:")
    print(value)

    # Must return the value since we took one in. Here we're not choosing
    # to mutate it, but we could.
    value
  },
  postserialize=function(res){
    print("We serialized the value as:")
    print(res$body)
  }
))

pr$handle("GET", "/", function(){ 123 })

## End(Not run)

## ------------------------------------------------
## Method `Plumber$handle`
## ------------------------------------------------

## Not run:
pr <- pr()
pr$handle("GET", "/", function(){
  "<html><h1>Programmatic Plumber!</h1></html>"
}, serializer=plumber::serializer_html())

## End(Not run)
```

```
## ------------------------------------------------
## Method `Plumber$setSerializer`
## ------------------------------------------------

## Not run:
pr <- pr()
pr$setSerializer(serializer_unboxed_json())

## End(Not run)

## ------------------------------------------------
## Method `Plumber$set404Handler`
## ------------------------------------------------

## Not run:
pr <- pr()
pr$set404Handler(function(req, res) {cat(req$PATH_INFO)})

## End(Not run)

## ------------------------------------------------
## Method `Plumber$setErrorHandler`
## ------------------------------------------------

## Not run:
pr <- pr()
pr$setErrorHandler(function(req, res, err) {
  message("Found error: ")
  str(err)
})

## End(Not run)
```

`PlumberEndpoint` *Plumber Endpoint*

Description

Plumber Endpoint

Plumber Endpoint

Details

Defines a terminal handler in a PLumber router.

Parameters values are obtained from parsing blocks of lines in a plumber file. They can also be provided manually for historical reasons.

Super classes

`plumber::Hookable` -> `plumber::PlumberStep` -> `PlumberEndpoint`

Public fields

`verbs` a character vector. http methods. For historical reasons we have to accept multiple verbs for a single path. Now it's simpler to just parse each separate verb/path into its own endpoint, so we just do that.

`path` a character string. endpoint path

`comments` endpoint comments

`responses` endpoint responses

`params` endpoint parameters

`tags` endpoint tags

`parsers` step allowed parsers

Methods

Public methods:

- `PlumberEndpoint$getTypedParams()`
- `PlumberEndpoint$canServe()`
- `PlumberEndpoint$matchesPath()`
- `PlumberEndpoint$new()`
- `PlumberEndpoint$getPathParams()`
- `PlumberEndpoint$getFuncParams()`
- `PlumberEndpoint$getEndpointParams()`
- `PlumberEndpoint$clone()`

Method `getTypedParams()`**:** retrieve endpoint typed parameters

Usage:

```
PlumberEndpoint$getTypedParams()
```

Method `canServe()`**:** ability to serve request

Usage:

```
PlumberEndpoint$canServe(req)
```

Arguments:

`req` a request object

Returns: a logical. `TRUE` when endpoint can serve request.

Method `matchesPath()`**:** determines if route matches requested path

Usage:

```
PlumberEndpoint$matchesPath(path)
```

Arguments:

`path` a url path

Returns: a logical. `TRUE` when endpoint matches the requested path.

Method `new()`**:** Create a new `PlumberEndpoint` object

Usage:

```
PlumberEndpoint$new(
  verbs,
  path,
  expr,
  envir,
  serializer,
  parsers,
  lines,
  params,
  comments,
  responses,
  tags
)
```

Arguments:

`verbs` Endpoint verb Ex: `"GET"`, `"POST"`

`path` Endpoint path. Ex: `"/index.html"`, `"/foo/bar/baz"`

`expr` Endpoint function or expression that evaluates to a function.

`envir` Endpoint environment

`serializer` Endpoint serializer. Ex: `serializer_json()`

`parsers` Can be one of:

- A `NULL` value
- A character vector of parser names
- A named `list()` whose keys are parser names names and values are arguments to be applied with `do.call()`
- A `TRUE` value, which will default to combining all parsers. This is great for seeing what is possible, but not great for security purposes

If the parser name `"all"` is found in any character value or list name, all remaining parsers will be added. When using a list, parser information already defined will maintain their existing argument values. All remaining parsers will use their default arguments.
Example:

```
# provide a character string
parsers = "json"

# provide a named list with no arguments
parsers = list(json = list())

# provide a named list with arguments; include `rds`
parsers = list(json = list(simplifyVector = FALSE), rds = list())

# default plumber parsers
parsers = c("json", "form", "text", "octet", "multi")
```

`lines` Endpoint block

`params` Endpoint params

`comments`, `responses`, `tags` Values to be used within the OpenAPI Spec

Returns: A new `PlumberEndpoint` object

Method `getPathParams()`**:** retrieve endpoint path parameters

Usage:

```
PlumberEndpoint$getPathParams(path)
```

Arguments:

`path` endpoint path

Method `getFuncParams()`**:** retrieve endpoint expression parameters

Usage:

```
PlumberEndpoint$getFuncParams()
```

Method `getEndpointParams()`**:** retrieve endpoint defined parameters

Usage:

```
PlumberEndpoint$getEndpointParams()
```

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`PlumberStatic` *Static file router*

Description

Static file router

Static file router

Details

Creates a router that is backed by a directory of files on disk.

Super classes

`plumber::Hookable` -> `plumber::Plumber` -> `PlumberStatic`

Methods

Public methods:

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

Method `new()`**:** Create a new `PlumberStatic` router

Usage:

```
PlumberStatic$new(direc, options)
```

Arguments:

`direc` a path to an asset directory.

`options` options to be evaluated in the `PlumberStatic` router environment

Returns: A new `PlumberStatic` router

Method `print()`**:** Print representation of `PlumberStatic()` router.

Usage:

```
PlumberStatic$print(prefix = "", topLevel = TRUE, ...)
```

Arguments:

`prefix` a character string. Prefix to append to representation.

`topLevel` a logical value. When method executed on top level router, set to `TRUE`.

`...` additional arguments for recursive calls

Returns: A terminal friendly representation of a `PlumberStatic()` router.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`PlumberStep` *plumber step R6 class*

Description

an object representing a step in the lifecycle of the treatment of a request by a plumber router.

Super class

`plumber::Hookable` -> `PlumberStep`

Public fields

`lines` lines from step block

`serializer` step serializer function

Methods

Public methods:

- `PlumberStep$new()`
- `PlumberStep$exec()`
- `PlumberStep$registerHook()`
- `PlumberStep$clone()`

Method `new()`**:** Create a new `PlumberStep()` object

Usage:

```
PlumberStep$new(expr, envir, lines, serializer)
```

Arguments:

`expr` step expr

`envir` step environment

`lines` step block

`serializer` step serializer

Returns: A new `PlumberStep` object

Method `exec()`**:** step execution function

Usage:

```
PlumberStep$exec(req, res)
```

Arguments:

`req, res` Request and response objects created by a Plumber request

Method `registerHook()`**:** step hook registration method

Usage:

```
PlumberStep$registerHook(
  stage = c("preexec", "postexec", "aroundexec"),
  handler
)
```

Arguments:

`stage` a character string.

`handler` a step handler function.

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

`plumb_api` *Process a Package's Plumber API*

Description

So that packages can ship multiple plumber routers, users should store their Plumber APIs in the `inst` subfolder `plumber` (`./inst/plumber/API_1/plumber.R`).

Usage

```
plumb_api(package = NULL, name = NULL)

available_apis(package = NULL)
```

Arguments

`package`	Package to inspect
`name`	Name of the package folder to `plumb()`.

Details

To view all available Plumber APIs across all packages, please call `available_apis()`. A `package` value may be provided to only display a particular package's Plumber APIs.

Value

A Plumber object. If either package or name is null, the appropriate available_apis() will be returned.

Functions

- plumb_api: plumb()s a package's Plumber API. Returns a Plumber router object
- available_apis: Displays all available package Plumber APIs. Returns a data.frame of package and name information.

---

pr *Create a new Plumber router*

Description

Create a new Plumber router

Usage

```
pr(
  file = NULL,
  filters = defaultPlumberFilters,
  envir = new.env(parent = .GlobalEnv)
)
```

Arguments

file	Path to file to plumb
filters	A list of Plumber filters
envir	An environment to be used as the enclosure for the routers execution

Value

A new Plumber router

Examples

```
## Not run:
pr() %>%
  pr_run()

## End(Not run)
```

`pr_cookie` *Store session data in encrypted cookies.*

Description

`plumber` uses the crypto R package `sodium`, to encrypt/decrypt `req$session` information for each server request.

Usage

```
pr_cookie(
  pr,
  key,
  name = "plumber",
  expiration = FALSE,
  http = TRUE,
  secure = FALSE,
  same_site = FALSE
)
```

Arguments

`pr`	A Plumber API. Note: The supplied Plumber API object will also be updated in place as well as returned by the function.
`key`	The secret key to use. This must be consistent across all R sessions where you want to save/restore encrypted cookies. It should be produced using `random_cookie_key`. Please see the "Storing secure keys" section for more details complex character string to bolster security.
`name`	The name of the cookie in the user's browser.
`expiration`	A number representing the number of seconds into the future before the cookie expires or a `POSIXt` date object of when the cookie expires. Defaults to the end of the user's browser session.
`http`	Boolean that adds the `HttpOnly` cookie flag that tells the browser to save the cookie and to NOT send it to client-side scripts. This mitigates cross-site scripting. Defaults to `TRUE`.
`secure`	Boolean that adds the `Secure` cookie flag. This should be set when the route is eventually delivered over HTTPS.
`same_site`	A character specifying the SameSite policy to attach to the cookie. If specified, one of the following values should be given: "Strict", "Lax", or "None". If "None" is specified, then the `secure` flag MUST also be set for the modern browsers to accept the cookie. An error will be returned if `same_site = "None"` and `secure = FALSE`. If not specified or a non-character is given, no SameSite policy is attached to the cookie.

Details

The cookie's secret encryption `key` value must be consistent to maintain `req$session` information between server restarts.

Storing secure keys

While it is very quick to get started with user session cookies using `plumber`, please exercise precaution when storing secure key information. If a malicious person were to gain access to the secret `key`, they would be able to eavesdrop on all `req$session` information and/or tamper with `req$session` information being processed.

Please:

- Do NOT store keys in source control.
- Do NOT store keys on disk with permissions that allow it to be accessed by everyone.
- Do NOT store keys in databases which can be queried by everyone.

Instead, please:

- Use a key management system, such as 'keyring' (preferred)
- Store the secret in a file on disk with appropriately secure permissions, such as "user read only" (`Sys.chmod("myfile.txt",mode = "0600")`), to prevent others from reading it.

Examples of both of these solutions are done in the Examples section.

See Also

- 'sodium': R bindings to 'libsodium'
- 'libsodium': A Modern and Easy-to-Use Crypto Library
- 'keyring': Access the system credential store from R
- Set-Cookie flags: Descriptions of different flags for `Set-Cookie`
- Cross-site scripting: A security exploit which allows an attacker to inject into a website malicious client-side code

Examples

```
## Not run:

## Set secret key using `keyring` (preferred method)
keyring::key_set_with_value("plumber_api", password = plumber::random_cookie_key())


pr() %>%
  pr_cookie(
    keyring::key_get("plumber_api"),
    name = "counter"
  ) %>%
  pr_get("/sessionCounter", function(req) {
    count <- 0
    if (!is.null(req$session$counter)){
      count <- as.numeric(req$session$counter)
    }
    req$session$counter <- count + 1
    return(paste0("This is visit #", count))
  }) %>%
  pr_run()


#### ------------------------------- ###
```

```
## Save key to a local file
pswd_file <- "normal_file.txt"
cat(plumber::random_cookie_key(), file = pswd_file)
# Make file read-only
Sys.chmod(pswd_file, mode = "0600")

pr() %>%
  pr_cookie(
    readLines(pswd_file, warn = FALSE),
    name = "counter"
  ) %>%
  pr_get("/sessionCounter", function(req) {
    count <- 0
    if (!is.null(req$session$counter)){
      count <- as.numeric(req$session$counter)
    }
    req$session$counter <- count + 1
    return(paste0("This is visit #", count))
  }) %>%
  pr_run()

## End(Not run)
```

`pr_filter` *Add a filter to Plumber router*

Description

Filters can be used to modify an incoming request, return an error, or return a response prior to the request reaching an endpoint.

Usage

```
pr_filter(pr, name, expr, serializer)
```

Arguments

`pr`	A Plumber API. Note: The supplied Plumber API object will also be updated in place as well as returned by the function.
`name`	A character string. Name of filter
`expr`	An expr that resolve to a filter function or a filter function
`serializer`	A serializer function

Value

The Plumber router with the defined filter added

Examples

```
## Not run:
pr() %>%
  pr_filter("foo", function(req, res) {
    print("This is filter foo")
    forward()
  }) %>%
  pr_get("/hi", function() "Hello") %>%
  pr_run()

## End(Not run)
```

`pr_handle` *Add handler to Plumber router*

Description

This collection of functions creates handlers for a Plumber router.

Usage

```
pr_handle(pr, methods, path, handler, preempt, serializer, endpoint, ...)

pr_get(pr, path, handler, preempt, serializer, endpoint, ...)

pr_post(pr, path, handler, preempt, serializer, endpoint, ...)

pr_put(pr, path, handler, preempt, serializer, endpoint, ...)

pr_delete(pr, path, handler, preempt, serializer, endpoint, ...)

pr_head(pr, path, handler, preempt, serializer, endpoint, ...)
```

Arguments

`pr`	A Plumber API. Note: The supplied Plumber API object will also be updated in place as well as returned by the function.
`methods`	Character vector of HTTP methods
`path`	The endpoint path
`handler`	A handler function
`preempt`	A preempt function
`serializer`	A Plumber serializer
`endpoint`	A `PlumberEndpoint` object
`...`	Additional arguments for `PlumberEndpoint`

Details

The generic pr_handle() creates a handle for the given method(s). Specific functions are implemented for the following HTTP methods:

- GET
- POST
- PUT
- DELETE
- HEAD Each function mutates the Plumber router in place, but also invisibly returns the updated router.

Value

A Plumber router with the handler added

Examples

```
## Not run:
pr() %>%
  pr_handle("GET", "/hi", function() "Hello World") %>%
  pr_run()

pr() %>%
  pr_handle(c("GET", "POST"), "/hi", function() "Hello World") %>%
  pr_run()

pr() %>%
  pr_get("/hi", function() "Hello World") %>%
  pr_post("/echo", function(req, res) {
    if (is.null(req$body)) return("No input")
    list(
      input = req$body
    )
  }) %>%
  pr_run()

## End(Not run)
```

---

pr_hook *Register a hook*

Description

Plumber routers support the notion of "hooks" that can be registered to execute some code at a particular point in the lifecycle of a request. Plumber routers currently support four hooks:

1. preroute(data,req,res)
2. postroute(data,req,res,value)
3. preserialize(data,req,res,value)

4. `postserialize(data,req,res,value)` In all of the above you have access to a disposable environment in the `data` parameter that is created as a temporary data store for each request. Hooks can store temporary data in these hooks that can be reused by other hooks processing this same request.

Usage

```
pr_hook(pr, stage, handler)

pr_hooks(pr, handlers)
```

Arguments

`pr`	A Plumber API. Note: The supplied Plumber API object will also be updated in place as well as returned by the function.
`stage`	A character string. Point in the lifecycle of a request.
`handler`	A hook function.
`handlers`	A named list of hook handlers

Details

One feature when defining hooks in Plumber routers is the ability to modify the returned value. The convention for such hooks is: any function that accepts a parameter named `value` is expected to return the new value. This could be an unmodified version of the value that was passed in, or it could be a mutated value. But in either case, if your hook accepts a parameter named `value`, whatever your hook returns will be used as the new value for the response.

You can add hooks using the `pr_hook`, or you can add multiple hooks at once using `pr_hooks`, which takes a named list in which the names are the names of the hooks, and the values are the handlers themselves.

Value

A Plumber router with the defined hook(s) added

Examples

```
## Not run:
pr() %>%
  pr_hook("preroute", function(req){
    cat("Routing a request for", req$PATH_INFO, "...\n")
  }) %>%
  pr_hooks(list(
    preserialize = function(req, value){
      print("About to serialize this value:")
      print(value)

      # Must return the value since we took one in. Here we're not choosing
      # to mutate it, but we could.
      value
    },
    postserialize = function(res){
      print("We serialized the value as:")
      print(res$body)
    }
```

```
  )) %>%
  pr_handle("GET", "/", function(){ 123 }) %>%
  pr_run()

## End(Not run)
```

`pr_mount` *Mount a Plumber router*

Description

Plumber routers can be "nested" by mounting one into another using the `mount()` method. This allows you to compartmentalize your API by paths which is a great technique for decomposing large APIs into smaller files. This function mutates the Plumber router (`pr()`) in place, but also invisibly returns the updated router.

Usage

```
pr_mount(pr, path, router)
```

Arguments

`pr`	The host Plumber router.
`path`	A character string. Where to mount router.
`router`	A Plumber router. Router to be mounted.

Value

A Plumber router with the supplied router mounted

Examples

```
## Not run:
pr1 <- pr() %>%
  pr_get("/hello", function() "Hello")

pr() %>%
  pr_get("/goodbye", function() "Goodbye") %>%
  pr_mount("/hi", pr1) %>%
  pr_run()

## End(Not run)
```

pr_run *Start a server using* plumber *object*

Description

port does not need to be explicitly assigned.

Usage

```
pr_run(pr, host = "127.0.0.1", port = getOption("plumber.port", NULL))
```

Arguments

pr	A Plumber API. Note: The supplied Plumber API object will also be updated in place as well as returned by the function.
host	A string that is a valid IPv4 or IPv6 address that is owned by this server, which the application will listen on. "0.0.0.0" represents all IPv4 addresses and "::/0" represents all IPv6 addresses.
port	A number or integer that indicates the server port that should be listened on. Note that on most Unix-like systems including Linux and Mac OS X, port numbers smaller than 1025 require root privileges.

Examples

```
## Not run:
pr() %>%
  pr_run()

pr() %>%
  pr_run(port = 5762, debug = TRUE)

## End(Not run)
```

pr_set_404 *Set the handler that is called when the incoming request can't be served*

Description

This function allows a custom error message to be returned when a request cannot be served by an existing endpoint or filter.

Usage

```
pr_set_404(pr, fun)
```

Arguments

pr	A Plumber API. Note: The supplied Plumber API object will also be updated in place as well as returned by the function.
fun	A handler function

Value

The Plumber router with a modified 404 handler

Examples

```
## Not run:
handler_404 <- function(req, res) {
  res$status <- 404
  res$body <- "Oops"
}

pr() %>%
  pr_get("/hi", function() "Hello") %>%
  pr_set_404(handler_404) %>%
  pr_run()

## End(Not run)
```

---

pr_set_api_spec *Set the OpenAPI Specification information*

Description

When set, it will be called with a character string corresponding to the API visual documentation url. This allows RStudio to open `swagger` docs when a Plumber router `pr_run()` method is executed using default `plumber.docs.callback` option.

Usage

```
pr_set_api_spec(pr, api)
```

Arguments

pr	A Plumber API. Note: The supplied Plumber API object will also be updated in place as well as returned by the function.
api	This can be • an OpenAPI Specification formatted list object • a function that accepts the OpenAPI Specification autogenerated by `plumber` and returns a OpenAPI Specification formatted list object. The value returned will not be validated for OAS compatibility.

Value

The Plumber router with the new OpenAPI Specification object or function.

Examples

```
## Not run:
# Set the API Spec to a function to use the auto-generated OAS object
pr() %>%
  pr_set_api_spec(function(spec) {
    spec$info$title <- Sys.time()
    spec
  }) %>%
  pr_get("/plus/<a:int>/<b:int>", function(a, b) { a + b }) %>%
  pr_run()

# Set the API Spec using an object
pr() %>%
  pr_set_api_spec(my_custom_object) %>%
  pr_get("/plus/<a:int>/<b:int>", function(a, b) { a + b }) %>%
  pr_run()

## End(Not run)
```

`pr_set_debug` *Set debug value to include error messages of routes cause an error*

Description

To hide any error messages in production, set the debug value to `FALSE`. The debug value is enabled by default for `interactive()` sessions.

Usage

```
pr_set_debug(pr, debug = interactive())
```

Arguments

pr	A Plumber API. Note: The supplied Plumber API object will also be updated in place as well as returned by the function.
debug	`TRUE` provides more insight into your API errors.

Value

The Plumber router with the new debug setting.

Examples

```
## Not run:
# Will contain the original error message
pr() %>%
  pr_set_debug(TRUE) %>%
  pr_get("/boom", function() stop("boom")) %>%
  pr_run()

# Will NOT contain an error message
pr() %>%
  pr_set_debug(FALSE) %>%
```

```
  pr_get("/boom", function() stop("boom")) %>%
  pr_run()

## End(Not run)
```

---

pr_set_docs *Set the API visual documentation*

Description

docs should be either a logical or a character value matching a registered visual documentation. When TRUE or a function, multiple handles will be added to Plumber object. OpenAPI json file will be served on paths /openapi.json and /swagger.json. Documentation will be served on paths /__docs__/index.html and /__docs__/. When using a function, it will receive the Plumber router as the first parameter and current OpenAPI Specification as the second. This function should return a list containing OpenAPI Specification. See http://spec.openapis.org/oas/v3.0.3

Usage

```
pr_set_docs(pr, docs = getOption("plumber.docs", TRUE), ...)
```

Arguments

pr	A Plumber API. Note: The supplied Plumber API object will also be updated in place as well as returned by the function.
docs	a character value or a logical value. If using options_plumber(), the value must be set before initializing your Plumber router.
...	Other params to be passed to the docs functions.

Value

The Plumber router with the new docs settings.

Examples

```
## Not run:
## View API using Swagger UI
# Official Website: https://swagger.io/tools/swagger-ui/
# install.packages("swagger")
if (require(swagger)) {
  pr() %>%
    pr_set_docs("swagger") %>%
    pr_get("/plus/<a:int>/<b:int>", function(a, b) { a + b }) %>%
    pr_run()
}

## View API using Redoc
# Official Website: https://github.com/Redocly/redoc
# remotes::install_github("https://github.com/meztez/redoc/")
if (require(redoc)) {
  pr() %>%
    pr_set_docs("redoc") %>%
    pr_get("/plus/<a:int>/<b:int>", function(a, b) { a + b }) %>%
```

```
    pr_run()
}

## View API using RapiDoc
# Official Website: https://github.com/mrin9/RapiDoc
# remotes::install_github("https://github.com/meztez/rapidoc/")
if (require(rapidoc)) {
  pr() %>%
    pr_set_docs("rapidoc") %>%
    pr_get("/plus/<a:int>/<b:int>", function(a, b) { a + b }) %>%
    pr_run()
}

## Disable the OpenAPI Spec UI
pr() %>%
  pr_set_docs(FALSE) %>%
  pr_get("/plus/<a:int>/<b:int>", function(a, b) { a + b }) %>%
  pr_run()

## End(Not run)
```

`pr_set_docs_callback` *Set the* `callback` *to tell where the API visual documentation is located*

Description

When set, it will be called with a character string corresponding to the API visual documentation url. This allows RStudio to open `swagger` docs when a Plumber router `pr_run()` method.

Usage

```
pr_set_docs_callback(pr, callback = getOption("plumber.docs.callback", NULL))
```

Arguments

`pr`	A Plumber API. Note: The supplied Plumber API object will also be updated in place as well as returned by the function.
`callback`	a callback function for taking action on the docs url.

Details

If using `options_plumber()`, the value must be set before initializing your Plumber router.

Value

The Plumber router with the new docs callback setting.

Examples

```
## Not run:
pr() %>%
  pr_set_docs_callback(function(url) { message("API location: ", url) }) %>%
  pr_get("/plus/<a:int>/<b:int>", function(a, b) { a + b }) %>%
  pr_run()

## End(Not run)
```

`pr_set_error` *Set the error handler that is invoked if any filter or endpoint generates an error*

Description

Set the error handler that is invoked if any filter or endpoint generates an error

Usage

```
pr_set_error(pr, fun)
```

Arguments

`pr`	A Plumber API. Note: The supplied Plumber API object will also be updated in place as well as returned by the function.
`fun`	An error handler function. This should accept `req`, `res`, and the error value

Value

The Plumber router with a modified error handler

Examples

```
## Not run:
handler_error <- function(req, res, err){
  res$status <- 500
  list(error = "Custom Error Message")
}

pr() %>%
  pr_get("/error", function() log("a")) %>%
  pr_set_error(handler_error) %>%
  pr_run()

## End(Not run)
```

`pr_set_parsers` *Set the default endpoint parsers for the router*

Description

By default, Plumber will parse JSON, text, query strings, octet streams, and multipart bodies. This function updates the default parsers for any endpoint that does not define their own parsers.

Usage

```
pr_set_parsers(pr, parsers)
```

Arguments

`pr`	A Plumber API. Note: The supplied Plumber API object will also be updated in place as well as returned by the function.

`parsers`	Can be one of:
	• A `NULL` value
	• A character vector of parser names
	• A named `list()` whose keys are parser names names and values are arguments to be applied with `do.call()`
	• A `TRUE` value, which will default to combining all parsers. This is great for seeing what is possible, but not great for security purposes
	If the parser name `"all"` is found in any character value or list name, all remaining parsers will be added. When using a list, parser information already defined will maintain their existing argument values. All remaining parsers will use their default arguments.
	Example:
	`# provide a character string`
	`parsers = "json"`
	
	`# provide a named list with no arguments`
	`parsers = list(json = list())`
	
	`# provide a named list with arguments; include \`rds\``
	`parsers = list(json = list(simplifyVector = FALSE), rds = list())`
	
	`# default plumber parsers`
	`parsers = c("json", "form", "text", "octet", "multi")`

Details

Note: The default set of parsers will be completely replaced if any value is supplied. Be sure to include all of your parsers that you would like to include.

Value

The Plumber router with the new default PlumberEndpoint parsers

pr_set_serializer *Set the default serializer of the router*

Description

By default, Plumber serializes responses to JSON. This function updates the default serializer to the function supplied via `serializer`

Usage

```
pr_set_serializer(pr, serializer)
```

Arguments

`pr`	A Plumber API. Note: The supplied Plumber API object will also be updated in place as well as returned by the function.
`serializer`	A serializer function

Value

The Plumber router with the new default serializer

pr_static *Add a static route to the* `plumber` *object*

Description

Add a static route to the `plumber` object

Usage

```
pr_static(pr, path, direc)
```

Arguments

`pr`	A Plumber API. Note: The supplied Plumber API object will also be updated in place as well as returned by the function.
`path`	The mounted path location of the static folder
`direc`	The local folder to be served statically

Examples

```
## Not run:
pr() %>%
  pr_static("/path", "./my_folder/location") %>%
  pr_run()

## End(Not run)
```

random_cookie_key *Random cookie key generator*

Description

Uses a cryptographically secure pseudorandom number generator from `sodium::helpers()` to generate a 64 digit hexadecimal string. 'sodium' wraps around 'libsodium'.

Usage

```
random_cookie_key()
```

Details

Please see `session_cookie` for more information on how to save the generated key.

Value

A 64 digit hexadecimal string to be used as a key for cookie encryption.

See Also

`session_cookie`

register_docs *Add visual documentation for plumber to use*

Description

`register_docs()` is used by other packages like `swagger`, `rapidoc`, and `redoc`. When you load these packages, it calls `register_docs()` to provide a user interface that can interpret your plumber OpenAPI Specifications.

Usage

```
register_docs(name, index, static = NULL)

registered_docs()
```

Arguments

`name`	Name of the visual documentation
`index`	A function that returns the HTML content of the landing page of the documentation. Parameters (besides `req` and `res`) will be supplied as if it is a regular `GET` route. Default parameter values may be used when setting the documentation `index` function. See the example below.
`static`	A function that returns the path to the static assets (images, javascript, css, fonts) the Docs will use.

Examples

```
## Not run:
# Example from the `swagger` R package
register_docs(
  name = "swagger",
  index = function(version = "3", ...) {
    swagger::swagger_spec(
      api_path = paste0(
        "window.location.origin + ",
        "window.location.pathname.replace(",
          "/\\(__docs__\\\\/|__docs__\\\\/index.html\\)$/, \"\"",
        ") + ",
        "\"openapi.json\""
      ),
      version = version
    )
  },
  static = function(version = "3", ...) {
    swagger::swagger_path(version)
  }
)

# When setting the docs, `index` and `static` function arguments can be supplied
# * via `pr_set_docs()`
# * or through URL query string variables
pr() %>%
  # Set default argument `version = 3` for the swagger `index` and `static` functions
  pr_set_docs("swagger", version = 3) %>%
  pr_get("/plus/<a:int>/<b:int>", function(a, b) { a + b }) %>%
  pr_run()

## End(Not run)
```

`register_parser` *Manage parsers*

Description

A parser is responsible for decoding the raw body content of a request into a list of arguments that can be mapped to endpoint function arguments. For instance, `parser_json()` parse content-type `application/json`.

Usage

```
register_parser(alias, parser, fixed = NULL, regex = NULL, verbose = TRUE)

registered_parsers()
```

Arguments

`alias`	An alias to map parser from the @parser plumber tag to the global parsers list.
`parser`	The parser function to be added. This build the parser function. See Details for more information.

fixed	A character vector of fixed string to be matched against a request content-type to use parser.
regex	A character vector of regex string to be matched against a request content-type to use parser.
verbose	Logical value which determines if a warning should be displayed when alias in map are overwritten.

Details

When parser is evaluated, it should return a parser function. Parser matching is done first by content-type header matching on fixed then by using a regular expressions on regex. Note that plumber strip the header from ; charset* to perform matching.

There is a special case when no content-type header is provided that will use a parser_json() when it detects a json string.

Functions signature should include value, ... and possibly content_type, filename. Other parameters may be provided if you want to use the headers from webutils::parse_multipart().

Parser function structure is something like below.

```
parser <- function(parser_arguments_here) {
  # return a function to parse a raw value
  function(value, ...) {
    # do something with raw value
  }
}
```

Functions

- registered_parsers: Return all registered parsers

Examples

```
# `content-type` header is mostly used to look up charset and adjust encoding
parser_dcf <- function(...) {
  function(value, content_type = "text/x-dcf", ...) {
    charset <- get_character_set(content_type)
    value <- rawToChar(value)
    Encoding(value) <- charset
    read.dcf(value, ...)
  }
}

# Could also leverage existing parsers
parser_dcf <- function(...) {
  parser_read_file(function(tmpfile) {
    read.dcf(tmpfile, ...)
  })
}

# Register the newly created parser
## Not run: register_parser("dcf", parser_dcf, fixed = "text/x-dcf")
```

`register_serializer` *Register a Serializer*

Description

A serializer is responsible for translating a generated R value into output that a remote user can understand. For instance, the `serializer_json` serializes R objects into JSON before returning them to the user. The list of available serializers in plumber is global.

Usage

```
register_serializer(name, serializer, verbose = TRUE)

registered_serializers()
```

Arguments

`name`	The name of the serializer (character string)
`serializer`	The serializer function to be added. This function should accept arguments that can be supplied when `plumb()`ing a file. This function should return a function that accepts four arguments: `value`, `req`, `res`, and `errorHandler`. See `print(serializer_json)` for an example.
`verbose`	Logical value which determines if a message should be printed when overwriting serializers

Details

There are three main building-block serializers:

- `serializer_headers`: the base building-block serializer that is required to have `as_attachment()` work
- `serializer_content_type()`: for setting the content type. (Calls `serializer_headers()`)
- `serializer_device()`: add endpoint hooks to turn a graphics device on and off in addition to setting the content type. (Uses `serializer_content_type()`)

Functions

- `register_serializer`: Register a serializer with a name
- `registered_serializers`: Return a list of all registered serializers

Examples

```
# `serializer_json()` calls `serializer_content_type()` and supplies a serialization function
print(serializer_json)
# serializer_content_type() calls `serializer_headers()` and supplies a serialization function
print(serializer_content_type)
```

`serializer_headers` *Plumber Serializers*

Description

Serializers are used in Plumber to transform the R object produced by a filter/endpoint into an HTTP response that can be returned to the client. See here for more details on Plumber serializers and how to customize their behavior.

Usage

```
serializer_headers(headers = list(), serialize_fn = identity)

serializer_content_type(type, serialize_fn = identity)

serializer_csv(..., type = "text/csv; charset=UTF-8")

serializer_tsv(..., type = "text/tab-separated-values; charset=UTF-8")

serializer_html(type = "text/html; charset=UTF-8")

serializer_json(..., type = "application/json")

serializer_unboxed_json(auto_unbox = TRUE, ..., type = "application/json")

serializer_rds(version = "2", ascii = FALSE, ..., type = "application/rds")

serializer_feather(type = "application/feather")

serializer_yaml(..., type = "text/x-yaml; charset=UTF-8")

serializer_text(
  ...,
  serialize_fn = as.character,
  type = "text/plain; charset=UTF-8"
)

serializer_format(..., type = "text/plain; charset=UTF-8")

serializer_print(..., type = "text/plain; charset=UTF-8")

serializer_cat(..., type = "text/plain; charset=UTF-8")

serializer_write_file(type, write_fn, fileext = NULL)

serializer_htmlwidget(..., type = "text/html; charset=UTF-8")

serializer_device(type, dev_on, dev_off = grDevices::dev.off)

serializer_jpeg(..., type = "image/jpeg")
```

```
serializer_png(..., type = "image/png")

serializer_svg(..., type = "image/svg+xml")

serializer_bmp(..., type = "image/bmp")

serializer_tiff(..., type = "image/tiff")

serializer_pdf(..., type = "application/pdf")
```

Arguments

headers	list() of headers to add to the response object
serialize_fn	Function to serialize the data. The result object will be converted to a character string. Ex: jsonlite::toJSON().
type	The value to provide for the Content-Type HTTP header.
...	extra arguments supplied to respective internal serialization function.
auto_unbox	automatically unbox all atomic vectors of length 1. It is usually safer to avoid this and instead use the unbox function to unbox individual elements. An exception is that objects of class AsIs (i.e. wrapped in I()) are not automatically unboxed. This is a way to mark single values as length-1 arrays.
version	the workspace format version to use. NULL specifies the current default version (3). The only other supported value is 2, the default from R 1.4.0 to R 3.5.0.
ascii	a logical. If TRUE or NA, an ASCII representation is written; otherwise (default) a binary one. See also the comments in the help for save.
write_fn	Function that should write serialized content to the temp file provided. write_fn should have the function signature of function(value,tmp_file){}.
fileext	A non-empty character vector giving the file extension. This value will try to be inferred from the content type provided.
dev_on	Function to turn on a graphics device. The graphics device dev_on function will receive any arguments supplied to the serializer in addition to filename. filename points to the temporary file name that should be used when saving content.
dev_off	Function to turn off the graphics device. Defaults to grDevices::dev.off()

Functions

- serializer_headers: Add a static list of headers to each return value. Will add Content-Disposition header if a value is the result of as_attachment().
- serializer_content_type: Adds a Content-Type header to the response object
- serializer_csv: CSV serializer. See also: readr::format_csv()
- serializer_tsv: TSV serializer. See also: readr::format_tsv()
- serializer_html: HTML serializer
- serializer_json: JSON serializer. See also: jsonlite::toJSON()
- serializer_unboxed_json: JSON serializer with auto_unbox defaulting to TRUE. See also: jsonlite::toJSON()
- serializer_rds: RDS serializer. See also: base::serialize()

- serializer_feather: feather serializer. See also: feather::write_feather()
- serializer_yaml: YAML serializer. See also: yaml::as.yaml()
- serializer_text: Text serializer. See also: as.character()
- serializer_format: Text serializer. See also: format()
- serializer_print: Text serializer. Captures the output of print()
- serializer_cat: Text serializer. Captures the output of cat()
- serializer_write_file: Write output to a temp file whose contents are read back as a serialized response. serializer_write_file() creates (and cleans up) a temp file, calls the serializer (which should write to the temp file), and then reads the contents back as the serialized value. If the content type starts with "text", the return result will be read into a character string, otherwise the result will be returned as a raw vector.
- serializer_htmlwidget: htmlwidget serializer. See also: htmlwidgets::saveWidget()
- serializer_device: Helper method to create graphics device serializers, such as serializer_png(). See also: endpoint_serializer()
- serializer_jpeg: JPEG image serializer. See also: grDevices::jpeg()
- serializer_png: PNG image serializer. See also: grDevices::png()
- serializer_svg: SVG image serializer. See also: grDevices::svg()
- serializer_bmp: BMP image serializer. See also: grDevices::bmp()
- serializer_tiff: TIFF image serializer. See also: grDevices::tiff()
- serializer_pdf: PDF image serializer. See also: grDevices::pdf()

`session_cookie` *Store session data in encrypted cookies.*

Description

plumber uses the crypto R package sodium, to encrypt/decrypt req$session information for each server request.

Usage

```
session_cookie(
  key,
  name = "plumber",
  expiration = FALSE,
  http = TRUE,
  secure = FALSE,
  same_site = FALSE
)
```

Arguments

key	The secret key to use. This must be consistent across all R sessions where you want to save/restore encrypted cookies. It should be produced using random_cookie_key. Please see the "Storing secure keys" section for more details complex character string to bolster security.
name	The name of the cookie in the user's browser.

`expiration`	A number representing the number of seconds into the future before the cookie expires or a `POSIXt` date object of when the cookie expires. Defaults to the end of the user's browser session.
`http`	Boolean that adds the `HttpOnly` cookie flag that tells the browser to save the cookie and to NOT send it to client-side scripts. This mitigates cross-site scripting. Defaults to `TRUE`.
`secure`	Boolean that adds the `Secure` cookie flag. This should be set when the route is eventually delivered over HTTPS.
`same_site`	A character specifying the SameSite policy to attach to the cookie. If specified, one of the following values should be given: "Strict", "Lax", or "None". If "None" is specified, then the `secure` flag MUST also be set for the modern browsers to accept the cookie. An error will be returned if `same_site = "None"` and `secure = FALSE`. If not specified or a non-character is given, no SameSite policy is attached to the cookie.

Details

The cookie's secret encryption `key` value must be consistent to maintain `req$session` information between server restarts.

Storing secure keys

While it is very quick to get started with user session cookies using `plumber`, please exercise precaution when storing secure key information. If a malicious person were to gain access to the secret `key`, they would be able to eavesdrop on all `req$session` information and/or tamper with `req$session` information being processed.

Please:

- Do NOT store keys in source control.
- Do NOT store keys on disk with permissions that allow it to be accessed by everyone.
- Do NOT store keys in databases which can be queried by everyone.

Instead, please:

- Use a key management system, such as 'keyring' (preferred)
- Store the secret in a file on disk with appropriately secure permissions, such as "user read only" (`Sys.chmod("myfile.txt",mode = "0600")`), to prevent others from reading it.

Examples of both of these solutions are done in the Examples section.

See Also

- 'sodium': R bindings to 'libsodium'
- 'libsodium': A Modern and Easy-to-Use Crypto Library
- 'keyring': Access the system credential store from R
- Set-Cookie flags: Descriptions of different flags for `Set-Cookie`
- Cross-site scripting: A security exploit which allows an attacker to inject into a website malicious client-side code

Examples

```
## Not run:

## Set secret key using `keyring` (preferred method)
keyring::key_set_with_value("plumber_api", plumber::random_cookie_key())


# Load a plumber API
plumb_api("plumber", "01-append") %>%
  # Add cookie support via `keyring`
  pr_cookie(
    keyring::key_get("plumber_api")
  ) %>%
  pr_run()


#### ------------------------------- ###


## Save key to a local file
pswd_file <- "normal_file.txt"
cat(plumber::random_cookie_key(), file = pswd_file)
# Make file read-only
Sys.chmod(pswd_file, mode = "0600")


# Load a plumber API
plumb_api("plumber", "01-append") %>%
  # Add cookie support and retrieve secret key from file
  pr_cookie(
    readLines(pswd_file, warn = FALSE)
  ) %>%
  pr_run()

## End(Not run)
```

`validate_api_spec` *Validate OpenAPI Spec*

Description

Validate an OpenAPI Spec using Swagger CLI which calls Swagger Parser.

Usage

```
validate_api_spec(pr, verbose = TRUE)
```

Arguments

`pr`	A Plumber API
`verbose`	Logical that determines if a "is valid" statement is displayed. Defaults to `TRUE`

Details

If the api is deemed invalid, an error will be thrown.

This function is VERY **Experimental** and may be altered, changed, or removed in the future.

Examples

```
## Not run:
pr <- plumb_api("plumber", "01-append")
validate_api_spec(pr)

## End(Not run)
```

Index