Package 'plumber2'

December 18, 2025

Title Easy and Powerful Webservers

Version 0.1.1

Description Automatically create a webserver from annotated 'R' files or by building it up programmatically. Provides automatic 'OpenAPI' documentation, input handling, async support, and middleware support.

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 7.3.3

Depends R (>= 4.1)

Imports base64enc, cli, fiery (>= 1.3.0), firesafety, firesale, firestorm, fs, jsonlite, promises, R6, ragg, rapidoc, readr, reqres (>= 1.0.0), rlang (>= 1.1.0), routr (>= 1.0.0), roxygen2, stringi, svglite, utils, webutils, yaml

Suggests arrow, callr, geojsonsf, htmlwidgets, later, mirai, nanoparquet, quarto, redoc, shiny, storr, swagger, testthat (>= 3.0.0)

VignetteBuilder quarto

URL https://plumber2.posit.co/, https://github.com/posit-dev/plumber2

BugReports https://github.com/posit-dev/plumber2/issues

Config/testthat/edition 3

NeedsCompilation no

Author Thomas Lin Pedersen [aut, cre] (ORCID: <https://orcid.org/0000-0002-5147-4711>), Posit Software, PBC [cph, fnd] (ROR: <https://ror.org/03wc8by49>)

Maintainer Thomas Lin Pedersen <thomas.pedersen@posit.co>

Repository CRAN

Date/Publication 2025-12-18 07:50:07 UTC

Contents

`add_plumber2_tag` *Add a tag extension to plumber2*

Description

Package authors can extend plumber2 with their own functionalities. If they wish to add a new tag to be used when writing annotated plumber2 routes they can use this function. If so, it should be called when the package is loaded.

Usage

```
add_plumber2_tag(tag, handler)
```

Arguments

tag	The name of the tag
handler	A handler function for the tag. See *Details*

Details

The handler argument must be a function with the arguments block, call, tags, values, and env. block is a list with the currently parsed information from the block. You can add or modify the values within to suit your need as well as subclass it. You should not remove any values as others might need them. call is the parsed value of whatever expression was beneath the plumber2 block. tags is a character vector of all the tags in the block, and values is a list of all the values associated with the tags (that is, whatever comes after the tag in the block). The values are unparsed. You should assume that all tags not relevant for your extension has already been handled and incorporated into block. The function must return a modified version of block. If you add a subclass to block you should make sure that a method for apply_plumber2_block() for the subclass exists.

Value

This function is called for its side effects

See Also

apply_plumber2_block()

Examples

```
# Add a tag that says hello when used
add_plumber2_tag("hello", function(block, call, tags, values, env) {
  message("Hello")
  class(block) <- c("hello_block", class(block))
  block
})
```

---

api *Create a new plumber API, optionally based on one or more plumber files*

Description

This is the main way to create a new Plumber2 object that encapsulates your full api. It is also possible to add files to the API after creation using api_parse()

Usage

```
api(
  ...,
  host = get_opts("host", "127.0.0.1"),
  port = get_opts("port", 8080),
  doc_type = get_opts("docType", "rapidoc"),
  doc_path = get_opts("docPath", "__docs__"),
  reject_missing_methods = get_opts("rejectMissingMethods", FALSE),
  ignore_trailing_slash = get_opts("ignoreTrailingSlash", TRUE),
  max_request_size = get_opts("maxRequestSize"),
  shared_secret = get_opts("sharedSecret"),
  compression_limit = get_opts("compressionLimit", 1000),
  default_async = get_opts("async", "mirai"),
  env = caller_env()
)

is_plumber_api(x)

api_parse(api, ...)
```

Arguments

`...`	plumber files or directories containing plumber files to be parsed in the given order. The order of parsing determines the final order of the routes in the stack. If `...` contains a `_server.yml` file then all other files in `...` will be ignored and the `_server.yml` file will be used as the basis for the API
`host`	A string that is a valid IPv4 address that is owned by this server
`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 macOS, port numbers smaller than 1024 require root privileges.
`doc_type`	The type of API documentation to generate. Can be either `"rapidoc"` (the default), `"redoc"`, `"swagger"`, or `NULL` (equating to not generating API docs)
`doc_path`	The URL path to serve the api documentation from
`reject_missing_methods`	Should requests to paths that doesn't have a handler for the specific method automatically be rejected with a 405 Method Not Allowed response with the correct Allow header informing the client of the implemented methods. Assigning a handler to `"any"` for the same path at a later point will overwrite this functionality. Be aware that setting this to `TRUE` will prevent the request from falling through to other routes that might have a matching method and path. This setting anly affects handlers on the request router.
`ignore_trailing_slash`	Logical. Should the trailing slash of a path be ignored when adding handlers and handling requests. Setting this will not change the request or the path associated with but just ensure that both `path/to/resource` and `path/to/resource/` ends up in the same handler.

`max_request_size`
: Sets a maximum size of request bodies. Setting this will add a handler to the header router that automatically rejects requests based on their `Content-Length` header

`shared_secret`
: Assigns a shared secret to the api. Setting this will add a handler to the header router that automatically rejects requests if their `Plumber-Shared-Secret` header doesn't contain the same value. Be aware that this type of authentication is very weak. Never put the shared secret in plain text but rely on e.g. the keyring package for storage. Even so, if requests are send over HTTP (not HTTPS) then anyone can read the secret and use it

`compression_limit`
: The size threshold in bytes for trying to compress the response body (it is still dependant on content negotiation)

`default_async`
: The default evaluator to use for async request handling

`env`
: The parent environment to the environment the files should be evaluated in. Each file will be evaluated in it's own environment so they don't interfere with each other

`x`
: An object to test for whether it is a plumber api

`api`
: A plumber2 api object to parse files into

Value

A Plumber2 object

See Also

`api_package()` for creating an api based on files distributed with a package

`get_opts()` for how to set default options

Examples

```
# When creating an API programmatically you'll usually initialise the object
# without pointing to any route files or a _server.yml file
pa <- api()

# You can pass it a directory and it will load up all recognised files it
# contains
example_dir <- system.file("plumber2", "quickstart", package = "plumber2")
pa <- api(example_dir)

# Or you can pass files directly
pa <- api(list.files(example_dir, full.names = TRUE)[1])
```

`api_add_route` *Add a new route to either the request or header router*

Description

This function allows explicit creation of routes or addition/merging of a predefined routr::Route into the router of the api. A new route can also be created with the `route` argument when adding a handler. However, that way will always add new routes to the end of the stack, whereas using `api_add_route()` allows you full control of the placement.

Usage

```
api_add_route(api, name, route = NULL, header = FALSE, after = NULL, root = "")
```

Arguments

`api`	A plumber2 api object to add the route to
`name`	The name of the route to add. If a route is already present with this name then the provided route (if any) is merged into it
`route`	The route to add. If `NULL` a new empty route will be created
`header`	Logical. Should the route be added to the header router?
`after`	The location to place the new route on the stack. `NULL` will place it at the end. Will not have an effect if a route with the given name already exists.
`root`	The root path to serve this route from.

Value

This functions return the `api` object allowing for easy chaining with the pipe

Using annotation

There is no direct equivalent to this when using annotated route files. However you can name your route in a file by adding `@routeName <name>` to the first block of the file like so.

```
#* @routeName my_route
NULL
```

All relevant blocks in the file will then be added to this route, even if the route already exist. In that way you can split the definition of a single route out among multiple files if needed.

Examples

```
# Add a new route and use it for a handler
api() |>
  api_add_route("logger_route") |>
  api_any(
    "/*",
    function() {
      cat("I just handled a request!")
    },
    route = "logger_route"
  )
```

`api_assets` *Serve resources from your file system*

Description

plumber2 provides two ways to serve files from your server. One (`api_assets`) goes through R and gives you all the power you expect to further modify and work with the response. The other (api_statics) never hits the R process and as a result is blazing fast. However this comes with the price of very limited freedom to modify the response or even do basic authentication. Each has their place.

Usage

```
api_assets(
  api,
  at,
  path,
  default_file = "index.html",
  default_ext = "html",
  finalize = NULL,
  continue = FALSE,
  route = NULL
)

api_statics(
  api,
  at,
  path,
  use_index = TRUE,
  fallthrough = FALSE,
  html_charset = "utf-8",
  headers = list(),
  validation = NULL,
  except = NULL
)
```

Arguments

api	A plumber2 api object to add the rossource serving to
at	The path to serve the resources from
path	The location on the file system to map at to
default_file	The default file to look for if the path does not map to a file directly (see Details)
default_ext	The default file extension to add to the file if a file cannot be found at the provided path and the path does not have an extension (see Details)
finalize	An optional function to run if a file is found. The function will receive the request as the first argument, the response as the second, and anything passed on through ... in the dispatch method. Any return value from the function is discarded. The function must accept ...
continue	A logical that should be returned if a file is found. Defaults to FALSE indicating that the response should be send unmodified.
route	The name of the route in the header router to add the asset route to. Defaults to the last route in the stack. If the route does not exist it will be created as the last route in the stack
use_index	Should an index.html file be served if present when a client requests the folder
fallthrough	Should requests that doesn't match a file enter the request loop or have a 404 response send directly
html_charset	The charset to report when serving html files
headers	A list of headers to add to the response. Will be combined with the global headers of the app
validation	An optional validation pattern. Presently, the only type of validation supported is an exact string match of a header. For example, if validation is '"abc" = "xyz"', then HTTP requests must have a header named abc (case-insensitive) with the value xyz (case-sensitive). If a request does not have a matching header, than httpuv will give a 403 Forbidden response. If the character(0) (the default), then no validation check will be performed.
except	One or more url paths that should be excluded from the route. Requests matching these will enter the standard router dispatch. The paths are interpreted as subpaths to at, e.g. the final path to exclude will be at+exclude (see example)

Value

These functions return the api object allowing for easy chaining with the pipe

Using annotation

When using annotated route files the functionality of api_assets() can be achieved like this:

```
#* @assets my_wd/ ./
NULL
```

When using annotated route files the functionality of api_statics() can be achieved like this:

```
#* @statics my_docs/ ~/
#* @except my_secret_folder/
NULL
```

Examples

```
# Add asset serving through routr route
api() |>
  api_assets("my_wd/", "./")

# Add asset serving directly
api() |>
  api_statics("my_docs", "~/", except = "my_secret_folder/")
```

`api_datastore` *Persistent server-side data storage*

Description

While using a session cookie is a convenient solution to persistent data storage between requests it has the downside of requiring the data to be passed back and forth between server and client at every exchange. This makes it impractical for all but the smallest snippets of data. An alternative strategy is to use server-side storage which this function facilitates. It uses the firesale plugin under the hood to provide a list-like interface to a storr-backed key-value store. storr in turn provides interfaces to a range of backends such as redis, LMDB, and databases supported by DBI. Further it provides simpler (but setup-free) solutions such as using an environment (obviously less persistent) or a folder of rds files.

Usage

```
api_datastore(
  api,
  driver,
  store_name = "datastore",
  gc_interval = 3600,
  max_age = gc_interval
)
```

Arguments

`api`	A plumber2 api object to add the datastore setup to
`driver`	A storr compatible driver that defines the backend of the datastore
`store_name`	The argument name under which the datastore will be available to the request handlers
`gc_interval`	The interval between running garbage collection on the backend
`max_age`	The time since last request to pass before a session store is cleared

Details

Once you turn the datastore on with this function your request handlers will gain access to a new argument (defaults to `datastore` but this can be changed with the `store_name` argument). The `datastore` argument will contain a list holding two elements: `global` and `session` which in turn will be list-like interfaces to the underlying key-value store. The `global` element access a store shared by all sessions whereas the `session` element is scoped to the current session. Depending on the value of `max_age` the session specific data is purged once a certain amount of time has passed since the last request from that session.

Value

These functions return the `api` object allowing for easy chaining with the pipe

Using annotation

Session cookie setup doesn't have a dedicated annotation tag, but you can set it up in a `@plumber` block

```
#* @plumber
function(api) {
  api |>
    api_datastore(storr::driver_dbi(...))
}
```

Examples

```
api() |>
  api_datastore(storr::driver_environment()) |>
  api_get("hello", function(datastore) {
    if (length(datastore$session) == 0) {
      datastore$global$count <- (datastore$global$count %||% 0) + 1
      datastore$session$not_first_visit <- TRUE
      paste0("Welcome. You are visitor #", datastore$global$count)
    } else {
      "Welcome back"
    }
  })
```

`api_docs` *Configure your API for serving documentation for itself*

Description

The OpenAPI standard offers a way to describe the various endpoints of your api in machine- and human-readable way. On top of this, various solutions have been build to generate online documentation of the API based on a provided OpenAPI spec. plumber2 offers support for RapiDoc, Redoc, and Swagger as a UI frontend for the documentation and will also generate the spec for you based

on the tags in parsed files. If you are creating your API programmatically or you wish to add to the autogenerated docs you can add docs manually, either when adding a handler (using the doc argument), or with the `api_doc_add()` function

Usage

```
api_doc_setting(api, doc_type, doc_path)

api_doc_add(api, doc, overwrite = FALSE, subset = NULL)
```

Arguments

`api`	A plumber2 api object to add docs or doc settings to
`doc_type`	The type of API documentation to generate. Can be either `"rapidoc"` (the default), `"redoc"`, `"swagger"`, or `NULL` (equating to not generating API docs)
`doc_path`	The URL path to serve the api documentation from
`doc`	A list with the OpenAPI documentation, usually constrcuted with one of the helper functions
`overwrite`	Logical. Should already existing documentation be removed or should it be merged together with `doc`
`subset`	A character vector giving the path to the subset of the docs to assign `doc` to

Value

These functions return the `api` object allowing for easy chaining with the pipe

Using annotation

When using annotated route files documentation is automatically generated based on the annotation. The following tags will contribute to documentation:

- `@title`
- `@description`
- `@details`
- `@tos`
- `@license`
- `@contact`
- `@tag`
- `@param`
- `@query`
- `@body`
- `@response`
- `@parsers`
- `@serializers`

Documentation is only generated for annotations related to global documentation (a block followed by the `"_API"` sentinel), request handlers (a block including one of `@get`, `@head`, `@post`, `@put`, `@delete`, `@connect`, `@options`, `@trace`, `@patch`, or `@any`), or report generation (a block including `@report`)

Examples

```
# Serve the docs from a different path
api() |>
  api_doc_setting(doc_path = "__man__")

# Add documentation to the api programmatically
api() |>
  api_doc_add(openapi(
    info = openapi_info(
      title = "My awesome api",
      version = "1.0.0"
    )
  ))

# Add documentation to a subset of the docs
api() |>
  api_doc_add(
    openapi_operation(
      summary = "Get the current date",
      responses = list(
        "200" = openapi_response(
          description = "Current Date",
          content = openapi_content(
            "text/plain" = openapi_schema(character())
          )
        )
      )
    ),
    subset = c("paths", "/date", "get")
  )
```

`api_forward` *Set up a plumber2 api to act as a reverse proxy*

Description

You can set up your plumber2 api to act as reverse proxy and forward all requests to a specific path (and it's subpaths) to a different URL. In contrast to `api_shiny()`, `api_forward()` is not responsible for launching whatever service is being proxied so this should be handled elsewhere. The `path` will be stripped from the request before being forwarded to the url, meaning that if you set up a proxy on `my/proxy/` to `http://example.com`, then a request for `my/proxy/user/thomas` will end at `http://example.com/user/thomas`. Proxying is most useful when forwarding to internal servers though you are free to forward to public URLs as well. However, for the later you'd usually use a redirect instead (via `api_redirect()`)

Usage

```
api_forward(api, path, url, except = NULL)
```

Arguments

api	A plumber2 api to add the shiny app to
path	The path to serve the shiny app from
url	The url to forward to
except	Subpaths to path that should be exempt from forwarding

Value

This functions return the api object allowing for easy chaining with the pipe

Using annotation

You can set up a reverse proxy in your annotated route file using the @forward tag

```
#* @forward /proxy http://127.0.0.1:56789
NULL
```

Examples

```
# Serve wikipedia directly from your app
api() |>
  api_forward("my_wiki/", "https://www.wikipedia.org")
```

---

api_logger *Set logging function and access log format for the API*

Description

plumber2 has a build-in logging facility that takes care of logging any conditions that are caught, as well as access logs. Further it is possible to log custom messages using the log() method on the api object. However, the actual logging is handled by a customizable function that can be set. You can read more about the logging infrastructure in the fiery documentation. plumber2 reexports the loggers provided by fiery so they are immediately available to the user.

Usage

```
api_logger(api, logger = NULL, access_log_format = NULL)

logger_null()

logger_console(format = "{time} - {event}: {message}")

logger_file(file, format = "{time} - {event}: {message}")

logger_logger(default_level = "INFO")
```

```
logger_switch(..., default = logger_null())

common_log_format

combined_log_format
```

Arguments

api	A plumber2 api object to set the logger on
logger	A logger function. If NULL then the current logger is kept
access_log_format	A glue string giving the format for the access logs. plumber2 (through fiery) provides the predefined common_log_format and combined_log_format, but you can easily create your own. See fiery::loggers for which variables the glue string has access to.
format	A glue string specifying the format of the log entry
file	A file or connection to write to
default_level	The log level to use for events that are not request, websocket, message, warning, or error
...	A named list of loggers to use for different events. The same semantics as switch is used so it is possible to let events *fall through* e.g. logger_switch(error =, warning = logger_file('errors.log')).
default	A catch-all logger for use with events not defined in ...

Using annotation

Logger setup doesn't have a dedicated annotation tag, but you can set it up in a @plumber block

```
#* @plumber
function(api) {
  api |>
    api_logger(logger = logger_null())
}
```

Examples

```
# Use a different access log format
api() |>
  api_logger(access_log_format = combined_log_format)

# Turn off logging
api() |>
  api_logger(logger_null())
```

`api_message` *Add a handler to a WebSocket message*

Description

WebSockets is a bidirectional communication channel that can be established at the request of the client. While websocket communication is not really part of a standard REST api, it has many uses and can easily be used together with one.

Usage

```
api_message(api, handler, async = NULL, then = NULL)
```

Arguments

`api`	A plumber2 api object to add the handler to
`handler`	A function conforming to the specifications laid out in Details
`async`	If `FALSE` create a regular handler. If `TRUE`, use the default async evaluator to create an async handler. If a string, the async evaluator registered to that name is used. If a function is provided then this is used as the async evaluator. See the *Async* section for more detail
`then`	A list of function to be called once the async handler is done. The functions will be chained using `promises::then()`. See the *Async* section for more detail

Details

A handler for a websocket message is much simpler than for requests in general since it doesn't have to concern itself with methods, paths, and responses. Any message handler registered will get called in sequence when a websocket message is recieved from a client. Still, a few expectations apply

Handler Arguments:

The handler can take any of the following arguments:

- `message`: Either a raw vector if the message recieved is in binary form or a single string, giving the message sent from the client
- `server`: The Plumber2 object representing your server implementation
- `client_id`: A string uniquely identifying the session the request comes from
- `request`: The request that was initially used to establish the websocket connection with the client as a reqres::Request object

Handler Return Value:

It is not expected that a websocket message sends a response and thus the handler is not required to do anything like that. However, if the handler returns either a raw vector or a single string it is taken as a signal to send this back to the client. Any other return value is silently ignored.

Value

This functions return the `api` object allowing for easy chaining with the pipe

Async

You can handle websocket messages asynchronously if needed. Like with request handlers you can either do it manually by creating and returning a promise inside the handler, or by letting plumber2 convert your handler to an async handler using the `async` argument. Due to the nature of promises a handler being converted to a promise can't take `request` and `server` arguments, so if you need to manipulate these you need to use `then` (more on this shortly). The same conventions about return value holds for async message handlers as for regular ones.

Async chaining:

Because you can't manipulate `request` or `server` in the async handler it may be needed to add operations to perform once the async handler has finished. This can be done through the `then` argument. This takes a list of functions to chain to the promise using `promises::then()`. Before the `then` chain is executed the return value of the async handler will be send back to the client if it is a string or a raw vector. Each `then` call will receive the same arguments as a standard message handler as well as `result` which will hold the return value of the previous handler in the chain. For the first `then` call `result` will be whatever the main async handler returned. The return value of the last call in the chain will be silently ignored.

Using annotation

A websocket message handler can be added to an API in an annotated route file by using the `@message` tag

```
#* @message
function(message) {
  if (message == "Hello") {
    return("Hello, you...")
  }
}
```

You can create async handlers with `then` chaining using annotation, through the `@async` and `@then` tags

```
#* @message
#* @async
function(message) {
  if (message == "Hello") {
    return("Hello, you...")
  }
}
#* @then
function(server) {
  server$log("message", "websocket message received")
}
```

Examples

```
api() |>
  api_message(
    function(message) {
      if (message == "Hello") {
        return("Hello, you...")
      }
    }
  )
```

`api_on` *Add a handler to an event*

Description

During the life cycle of a plumber API various events will be fired, either automatically or manually. See the article on events in fiery for a full overview. `api_on()` allows you to add handlers that are called when specific events fire. `api_off()` can be used to remove the handler if necessary

Usage

```
api_on(api, event, handler, id = NULL)

api_off(api, id)
```

Arguments

`api`	A plumber2 api object to launch or stop
`event`	A string naming the event to listen for
`handler`	A function to call when `event` fires
`id`	A string uniquely identifying the handler. If `NULL` a random id will be generated making it impossible to remove the handler again

Value

These functions return the `api` object allowing for easy chaining with the pipe

Using annotation

Event handler setup doesn't have a dedicated annotation tag, but you can set it up in a `@plumber` block

```
#* @plumber
function(api) {
  api |>
    api_on("cycle-end", function(server) {
```

```
      server$log("message", "tick-tock")
    })
}
```

Examples

```
# Add a small console log to show the api is alive
pa <- api() |>
  api_on("cycle-end", function(server) {
    server$log("message", "tick-tock")
  }, id = "lifesign")

# Remove it again
pa |>
  api_off("lifesign")
```

`api_package` *Load up an API distributed with a package*

Description

Packages can included one or more api specification(s) by storing the annotated route files and/or `_server.yml` file in subfolders of `./inst/plumber2`. The name of the subfolder will be the name of the api

Usage

```
api_package(package = NULL, name = NULL, ...)
```

Arguments

`package`	The name of the package that provides the api. If `NULL` then a list of available apis across all installed packages is returned
`name`	The name of the api. If `NULL` then a list of available apis in the given package is returned
`...`	Arguments passed on to `api` `host` A string that is a valid IPv4 address that is owned by this server `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 macOS, port numbers smaller than 1024 require root privileges. `doc_type` The type of API documentation to generate. Can be either `"rapidoc"` (the default), `"redoc"`, `"swagger"`, or `NULL` (equating to not generating API docs) `doc_path` The URL path to serve the api documentation from

`reject_missing_methods` Should requests to paths that doesn't have a handler for the specific method automatically be rejected with a 405 Method Not Allowed response with the correct Allow header informing the client of the implemented methods. Assigning a handler to `"any"` for the same path at a later point will overwrite this functionality. Be aware that setting this to `TRUE` will prevent the request from falling through to other routes that might have a matching method and path. This setting anly affects handlers on the request router.

`ignore_trailing_slash` Logical. Should the trailing slash of a path be ignored when adding handlers and handling requests. Setting this will not change the request or the path associated with but just ensure that both `path/to/resource` and `path/to/resource/` ends up in the same handler.

`max_request_size` Sets a maximum size of request bodies. Setting this will add a handler to the header router that automatically rejects requests based on their `Content-Length` header

`shared_secret` Assigns a shared secret to the api. Setting this will add a handler to the header router that automatically rejects requests if their `Plumber-Shared-Secret` header doesn't contain the same value. Be aware that this type of authentication is very weak. Never put the shared secret in plain text but rely on e.g. the keyring package for storage. Even so, if requests are send over HTTP (not HTTPS) then anyone can read the secret and use it

`compression_limit` The size threshold in bytes for trying to compress the response body (it is still dependant on content negotiation)

`default_async` The default evaluator to use for async request handling

`env` The parent environment to the environment the files should be evaluated in. Each file will be evaluated in it's own environment so they don't interfere with each other

Value

If `package` or `name` is `NULL` then a data frame providing available apis filtered on either package or name (if any is provided) is returned. Otherwise a Plumber2 object representing the api is returned

Examples

```
# Load one of the plumber2 examples
api_package("plumber2", "quickstart")

# List all available apis
api_package()
```

`api_redirect` *Redirect request to another resource*

Description

While it is optimal that an API remains stable over its lifetime it is often not fully attainable. In order to direct requests for resources that has been moved to the new location you can add a redirect that ensures a smooth transition for clients still using the old path. Depending on the value of `permanent` the redirect will respond with a `307 Temporary Redirect` or `308 Permanent Redirect`. `from` and `to` can contain path parameters and wildcards which will be matched between the two to construct the correct redirect path. Further, `to` can either be a path to the same server or a fully qualified URL to redirect requests to another server alltogether.

Usage

```
api_redirect(api, method, from, to, permanent = TRUE)
```

Arguments

`api`	A plumber2 api object to add the redirect to
`method`	The HTTP method the redirect should respond to
`from`	The path the redirect should respond to
`to`	The path/URL to redirect the incoming request towards. After resolving any path parameters and wildcards it will be used in the `Location` header
`permanent`	Logical. Is the redirect considered permanent or temporary? Determines the type of redirct status code to use

Value

This functions return the `api` object allowing for easy chaining with the pipe

Using annotation

You can specify redirects in an annotated plumber file using the `@redirect` tag. Preceed the method with a `!` to mark the redirect as permanent

```
#* @redirect !get /old/data/* /new/data/*
#* @redirect any /unstable/endpoint /stable/endpoint
NULL
```

Examples

```
api() |>
  api_redirect("get", "/old/data/*", "/new/data/*")
```

`api_request_handlers` *Add a handler for a request*

Description

This family of functions facilitates adding a request handler for a specific HTTP method and path.

Usage

```
api_get(
  api,
  path,
  handler,
  serializers = NULL,
  parsers = NULL,
  use_strict_serializer = FALSE,
  download = FALSE,
  async = FALSE,
  then = NULL,
  doc = NULL,
  route = NULL
)

api_head(
  api,
  path,
  handler,
  serializers = NULL,
  parsers = NULL,
  use_strict_serializer = FALSE,
  download = FALSE,
  async = FALSE,
  then = NULL,
  doc = NULL,
  route = NULL
)

api_post(
  api,
  path,
  handler,
  serializers = NULL,
  parsers = NULL,
  use_strict_serializer = FALSE,
  download = FALSE,
  async = FALSE,
  then = NULL,
```

```
  doc = NULL,
  route = NULL
)

api_put(
  api,
  path,
  handler,
  serializers = NULL,
  parsers = NULL,
  use_strict_serializer = FALSE,
  download = FALSE,
  async = FALSE,
  then = NULL,
  doc = NULL,
  route = NULL
)

api_delete(
  api,
  path,
  handler,
  serializers = NULL,
  parsers = NULL,
  use_strict_serializer = FALSE,
  download = FALSE,
  async = FALSE,
  then = NULL,
  doc = NULL,
  route = NULL
)

api_connect(
  api,
  path,
  handler,
  serializers = NULL,
  parsers = NULL,
  use_strict_serializer = FALSE,
  download = FALSE,
  async = FALSE,
  then = NULL,
  doc = NULL,
  route = NULL
)

api_options(
  api,
```

```
  path,
  handler,
  serializers = NULL,
  parsers = NULL,
  use_strict_serializer = FALSE,
  download = FALSE,
  async = FALSE,
  then = NULL,
  doc = NULL,
  route = NULL
)

api_trace(
  api,
  path,
  handler,
  serializers = NULL,
  parsers = NULL,
  use_strict_serializer = FALSE,
  download = FALSE,
  async = FALSE,
  then = NULL,
  doc = NULL,
  route = NULL
)

api_patch(
  api,
  path,
  handler,
  serializers = NULL,
  parsers = NULL,
  use_strict_serializer = FALSE,
  download = FALSE,
  async = FALSE,
  then = NULL,
  doc = NULL,
  route = NULL
)

api_any(
  api,
  path,
  handler,
  serializers = NULL,
  parsers = NULL,
  use_strict_serializer = FALSE,
  download = FALSE,
```

```
  async = FALSE,
  then = NULL,
  doc = NULL,
  route = NULL
)
```

Arguments

api	A plumber2 api object to add the handler to
path	A string giving the path the handler responds to. See Details
handler	A handler function to call when a request is matched to the path
serializers	A named list of serializers that can be used to format the response before sending it back to the client. Which one is selected is based on the request `Accept` header. See `get_serializers()` for a helper to construct this
parsers	A named list of parsers that can be used to parse the request body before passing it in as the `body` argument. Which one is selected is based on the request `Content-Type` header. See `get_parsers()` for a helper to construct this
use_strict_serializer	By default, if a serializer that respects the requests `Accept` header cannot be found, then the first of the provided ones are used. Setting this to `TRUE` will instead send back a `406 Not Acceptable` response
download	Should the response mark itself for download instead of being shown inline? Setting this to `TRUE` will set the `Content-Disposition` header in the response to `attachment`. Setting it to a string is equivalent to setting it to `TRUE` but will in addition also set the default filename of the download to the string value
async	If `FALSE` create a regular handler. If `TRUE`, use the default async evaluator to create an async handler. If a string, the async evaluator registered to that name is used. If a function is provided then this is used as the async evaluator. See the *Async* section for more detail
then	A list of function to be called once the async handler is done. The functions will be chained using `promises::then()`. See the *Async* section for more detail
doc	A list with the OpenAPI spec for the endpoint
route	The route this handler should be added to. Defaults to the last route in the stack. If the route does not exist it will be created as the last route in the stack

Value

These functions return the `api` object allowing for easy chaining with the pipe

Using annotation

Handlers can be specified in an annotated route file using one of the method tags followed by the path it pertains to. You can use various tags to descripe the handler and these will automatically be converted to OpenAPI documentation. Further, additional tags allow you to modify the behaviour of the handler, reflecting the arguments available in the functional approach.

```
#* A handler for /user/<username>
#*
#* @param username:string The name of the user to provide information on
#*
#* @get /user/<username>
#*
#* @response 200:{name:string, age:integer, hobbies:[string]} Important
#* information about the user such as their name, age, and hobbies
#*
function(username) {
  find_user_in_db(username)
}
```

Handlers can be specified in an annotated route file using one of the method tags followed by the path it pertains to. You can use various tags to describe the handler and these will automatically be converted to OpenAPI documentation. Further, additional tags allow you to modify the behaviour of the handler, reflecting the arguments available in the functional approach.

```
#* A handler for /user/<username>
#*
#* @param username:string The name of the user to provide information on
#*
#* @get /user/<username>
#*
#* @response 200:{name:string, age:integer, hobbies:[string]} Important
#* information about the user such as their name, age, and hobbies
#*
function(username) {
  find_user_in_db(username)
}
```

You can create async handlers with then chaining using annotation, through the `@async` and `@then` tags

```
#* A handler for /user/<username>
#*
#* @param username:string The name of the user to provide information on
#*
#* @get /user/<username>
#*
#* @response 200:{name:string, age:integer, hobbies:[string]} Important
#* information about the user such as their name, age, and hobbies
#*
#* @async
function(username) {
  find_user_in_db(username)
}
#* @then
```

```
function(server, response) {
  server$log("message", "async operation completed")
  response$set_header("etag", "abcdef")
  Next
}
```

HTTP Methods

The HTTP specs provide a selection of specific methods that clients can send to the server (your plumber api). While there is no enforcement that the server follows any conventions you should strive to create a server API that adheres to common expectations. It is not required that a server understands all methods, most often the opposite is true. The HTTP methods are described below, but consider consulting MDN to get acquainted with the HTTP spec in general

- `GET`: This method is used to request specific content and is perhaps the most ubiquitous method in use. `GET` requests should only retrieve data and should not contain any body content
- `HEAD`: This method is identical to `GET`, except the response should only contain headers, no body. Apart from this it is expected that a `HEAD` request is identical to a `GET` request for the same resource
- `POST`: This method delivers content, in the form of a request body, to the server, potentially causing a change in the server. In the context of plumber2 it is often used to call functions that require input larger than what can be put in the URL
- `PUT`: This method is used to update a specific resource on the server. In the context of a standard plumber2 server this is rarely relevant, though usage can come up. `PUT` is considered by clients to be indemptotent meaning that sending the same `PUT` request multiple times have no effect
- `DELETE`: This method deletes a resource and is the opposite to `PUT`. As with `PUT` this method has limited use in most standard plumber2 servers
- `CONNECT`: This method request the establishment of a proxy tunnel. It is considered advanced use and is very unlikely to have a usecase for your plumber2 api
- `OPTIONS`: This method is used by clients to query a server about what methods and other settings are supported on a server
- `TRACE`: This method is a form of ping that should send a response containing the request (stripped of any sensitive information). Many servers disallow this method due to security concerns
- `PATCH`: This method is like `PUT` but allows partial modification of a resource

Apart from the above, plumber2 also understands the `ANY` method which responds to requests to any of the above methods, assuming that a specific handler for the method is not found. As the semantics of the various methods are quite different an `ANY` handler should mainly be used for rejections or for setting specific broad headers on the response, not as the main handler for the request

The Path

The path defines the URL the request is being made to with the root removed. If your plumber2 server runs from `http://example.com/api/` and a request is made to `http://example.com/api/user/thomas/`, then the path would be `user/thomas/`. Paths can be static like the prior example, or dynamic as described below:

Path arguments:

Consider you have a bunch of users. It would be impractical to register a handler for each one of them. Instead you can use a dynamic path like with the following syntax: `user/<username>/`. This path would be matched to any requests made to `user/..something../`. The actual value of `..something..` (e.g. `thomas`) would be made available to the handler (see below). A path can contain multiple arguments if needed, such as `user/<username>/settings/<setting>/`

Path wildcards:

Apart from path arguments it is also possible to be even less specific by adding a wildcard to the path. The path `user/*` will match both `user/thomas/`, `user/thomas/settings/interests/`, and anything other path that begins with `user/`. As with arguments a path can contain multiple wildcards but the use of these have very diminishing returns. Contrary to path arguments the value(s) corresponding to `*` is not made available to the handler.

Path Priority:

With the existence of path arguments and wildcards it is possible that multiple handlers in a route can be matched to a single request. Since only one can be selected we need to determine which one wins. The priority is based on the specificity of the path. Consider a server containing the following handler paths: `user/thomas/`, `user/<username>/`, `user/<username>/settings/<setting>/`, `user/*`. These paths will have the following priority:

1. `user/<username>/settings/<setting>/`
2. `user/thomas/`
3. `user/<username>/`
4. `user/*`

The first spot is due to the fact that it is the path with the most elements so it is deemed most specific. For the remaining 3 they all have the same number of elements, but static paths are considered more specific than dynamic paths, and path arguments are considered more specific than wildcards.

A request made to `user/carl` will thus end up in the third handler, while a request made to `user/thomas` will end up in the second. This ordering makes it possible to both provide default handlers as well as specialisations for specific paths.

The Handler

The handler is a standard R function that is called when a request is made that matches the handlers path (unless a more specific handler path exists — see above). A handler function can perform any operation a normal R function can do, though you should consider strongly the security implications of your handler functions. However, there are certain expectations in plumber around the arguments a handler function takes and the return value it provides

Handler Arguments:

The handler function can take one or more of the following arguments.

- **Path arguments**: Any path arguments are passed on to the handler. If a handler is registered for the following path `user/<username>/settings/<setting>/` and it handles a request to `user/thomas/settings/interests/` then it will be called with `username = "thomas", setting = "interest"`
- `request`: The request the handler is responding to as a reqres::Request object
- `response`: The response being returned to the client as a reqres::Response object

- `server`: The Plumber2 object representing your server implementation
- `client_id`: A string uniquely identifying the session the request comes from
- `query`: A list giving any additional arguments passed into the handler as part of the url query string
- `body`: The request body, parsed as specified by the provided parsers

Handler Return Value:

Handlers can return a range of different value types, which will inform plumber2 what to do next:

Returning `Next` *or* `Break`*:*
These two control objects informs plumber2 to either proceed handling the request (`Next`) or return the response as is, circumventing any remaining routes (`Break`)

Returning `NULL` *or the* `response` *object:*
This is the same as returning `Next`, i.e. it signals that handling can proceed

Returning a ggplot2 object:
If you return a ggplot2 object it will get plotted for you (and added to the response assuming a graphics serializer is provided) before handling continues

Returning any other value:
Any kind of value returned that is not captured by the above description will be set to the response body (overwriting what was already there) and handling is then allowed to continue

Handler conditions:

Like any function in R, a handler may need to signal that something happened, either by throwing an error or warning or by emitting a message. You can use `stop()`, `warning()`, and `message()` as you are used to. For all of them, the condition message will end up in the log. Further, for `stop()` any further handling of the request will end and a `500 Internal Error` response is returned. To take more control over problems you can use the `abort_*()` family of conditions from reqres. Like `stop()` they will halt any further processing, but they also allow control over what kind of response is sent back, what kind of information about the issue is communicated to the client, and what kind of information is logged internally. The response they send back (except for `abort_status()`) all adhere to the HTTP Problem spec defined in RFC 9457.

While it may feel like a good idea to send a detailed error message back to the client it is often better to only inform the client of what they need to change to solve the issue. Too much information about internal implementation details can be a security risk and forwarding internal errors to a client can help inform the client about how the server has been implemented.

Async handling

plumber2 supports async handling of requests in one of two ways:

1. The handler you provide returns a promise object
2. You set `async = TRUE` (or the name of a registered async evaluator) when adding the handler

For 1), there is no more to do. You have full custody over the created promise and any `then()`-chaining that might be added to it. For 2) it is a bit different. In that case you provide a regular function and plumber2 takes care of converting it to a promise. Due to the nature of promises a handler being converted to a promise can't take `request`, `response`, and `server` arguments, so if you need to manipulate these you need to use `then` (more on this shortly). The async handler should yield the value that the response should ultimately get assigned to the body or have plotting side effects (in which case the plot will get added to the response).

Async chaining:

Because you can't manipulate `request` `response`, or `server` in the async handler it may be needed to add operations to perform once the async handler has finished. This can be done through the `then` argument (or using the `@then` tag in annotated route files). This takes a list of functions to chain to the promise using `promises::then()`. Before the `then` chain is executed the response will get the return value of the main handler asigned to the body. Each `then` call will receive the same arguments as a standard request handler as well as `result` which will hold the return value of the previous handler in the chain. For the first `then` call `result` will be a boolean signalling if the async handler wants request handling to proceed to the next route or terminate early. The last call in the chain must return `Next` or `Break` to signal if processing should be allowed to continue to the next route.

See Also

Other Request Handlers: `api_request_header_handlers`

Examples

```
# Standard use
api() |>
  api_get("/hello/<name:string>", function(name) {
    list(
      msg = paste0("Hello ", name, "!")
    )
  })

# Specify serializers
api() |>
  api_get(
    "/hello/<name:string>",
    function(name) {
      list(
        msg = paste0("Hello ", name, "!")
      )
    },
    serializers = get_serializers(c("json", "xml"))
  )

# Request a download and make it async
api() |>
  api_get(
    "/the_plot",
    function() {
      plot(1:10, 1:10)
    },
    serializers = get_serializers(c("png", "jpeg")),
    download = TRUE,
    async = TRUE
  )
```

`api_request_header_handlers` *Add a handler for a request header*

Description

These handlers are called before the request body has been recieved and lets you preemptively reject requests before recieving their full content. If the handler does not return Next then the request will be returned at once. Most of your logic, however, will be in the main handlers and you are asked to consult the api_request_handlers docs for in-depth details on how to use request handlers in general.

Usage

```
api_get_header(
  api,
  path,
  handler,
  serializers = NULL,
  parsers = NULL,
  use_strict_serializer = FALSE,
  download = FALSE,
  async = FALSE,
  then = NULL,
  route = NULL
)

api_head_header(
  api,
  path,
  handler,
  serializers = NULL,
  parsers = NULL,
  use_strict_serializer = FALSE,
  download = FALSE,
  async = FALSE,
  then = NULL,
  route = NULL
)

api_post_header(
  api,
  path,
  handler,
  serializers = NULL,
  parsers = NULL,
  use_strict_serializer = FALSE,
  download = FALSE,
```

```
  async = FALSE,
  then = NULL,
  route = NULL
)

api_put_header(
  api,
  path,
  handler,
  serializers = NULL,
  parsers = NULL,
  use_strict_serializer = FALSE,
  download = FALSE,
  async = FALSE,
  then = NULL,
  route = NULL
)

api_delete_header(
  api,
  path,
  handler,
  serializers = NULL,
  parsers = NULL,
  use_strict_serializer = FALSE,
  download = FALSE,
  async = FALSE,
  then = NULL,
  route = NULL
)

api_connect_header(
  api,
  path,
  handler,
  serializers = NULL,
  parsers = NULL,
  use_strict_serializer = FALSE,
  download = FALSE,
  async = FALSE,
  then = NULL,
  route = NULL
)

api_options_header(
  api,
  path,
  handler,
```

```
  serializers = NULL,
  parsers = NULL,
  use_strict_serializer = FALSE,
  download = FALSE,
  async = FALSE,
  then = NULL,
  route = NULL
)

api_trace_header(
  api,
  path,
  handler,
  serializers = NULL,
  parsers = NULL,
  use_strict_serializer = FALSE,
  download = FALSE,
  async = FALSE,
  then = NULL,
  route = NULL
)

api_patch_header(
  api,
  path,
  handler,
  serializers = NULL,
  parsers = NULL,
  use_strict_serializer = FALSE,
  download = FALSE,
  async = FALSE,
  then = NULL,
  route = NULL
)

api_any_header(
  api,
  path,
  handler,
  serializers = NULL,
  parsers = NULL,
  use_strict_serializer = FALSE,
  download = FALSE,
  async = FALSE,
  then = NULL,
  route = NULL
)
```

Arguments

api	A plumber2 api object to add the handler to
path	A string giving the path the handler responds to. See Details
handler	A handler function to call when a request is matched to the path
serializers	A named list of serializers that can be used to format the response before sending it back to the client. Which one is selected is based on the request Accept header. See get_serializers() for a helper to construct this
parsers	A named list of parsers that can be used to parse the request body before passing it in as the body argument. Which one is selected is based on the request Content-Type header. See get_parsers() for a helper to construct this
use_strict_serializer	By default, if a serializer that respects the requests Accept header cannot be found, then the first of the provided ones are used. Setting this to TRUE will instead send back a 406 Not Acceptable response
download	Should the response mark itself for download instead of being shown inline? Setting this to TRUE will set the Content-Disposition header in the response to attachment. Setting it to a string is equivalent to setting it to TRUE but will in addition also set the default filename of the download to the string value
async	If FALSE create a regular handler. If TRUE, use the default async evaluator to create an async handler. If a string, the async evaluator registered to that name is used. If a function is provided then this is used as the async evaluator. See the *Async* section for more detail
then	A list of function to be called once the async handler is done. The functions will be chained using promises::then(). See the *Async* section for more detail
route	The route this handler should be added to. Defaults to the last route in the stack. If the route does not exist it will be created as the last route in the stack

Value

These functions return the api object allowing for easy chaining with the pipe

Using annotation

Adding request header handler is done in the same way as for standard request handlers. The only difference is that you include a @header tag as well. It is not normal to document header requests as they usually exist as internal controls. You can add @noDoc to avoid generating OpenAPI docs for the handler

```
#* A header handler authorizing users
#*
#* @get /*
#*
#* @header
#* @noDoc
function(client_id, response) {
  if (user_is_allowed(username)) {
```

```
    Next
  } else {
    response$status <- 404L
    Break
  }
}
```

See Also

Other Request Handlers: api_request_handlers

Examples

```
# Simple size limit (better to use build-in functionality)
api() |>
  api_post_header(
    "/*",
    function(request, response) {
      if (request$get_header("content-type") > 1024) {
        response$status <- 413L
        Break
      } else {
        Next
      }
    }
  )
```

---

api_run *Launch the API*

Description

This function starts the api with the settings it has defined.

Usage

```
api_run(
  api,
  host = NULL,
  port = NULL,
  block = !is_interactive(),
  showcase = is_interactive(),
  ...,
  silent = FALSE
)

api_stop(api)
```

Arguments

api	A plumber2 api object to launch or stop
host, port	Host and port to run the api on. If not provided the host and port used during the creation of the Plumber2 api will be used
block	Should the console be blocked while running (alternative is to run in the background). Defaults to FALSE in interactive sessions and TRUE otherwise.
showcase	Should the default browser open up at the server address. If TRUE then a browser opens at the root of the api, unless the api contains OpenAPI documentation in which case it will open at that location. If a string the string is used as a path to add to the root before opening.
...	Arguments passed on to the start handler
silent	Should startup messaging by silenced

Value

These functions return the api object allowing for easy chaining with the pipe, even though they will often be the last part of the chain

Examples

```
pa <- api() |>
  api_get("/", function() {
    list(msg = "Hello World")
  }) |>
  api_on("start", function(...) {
    cat("I'm alive")
  })

# Start the server
pa |> api_run(block = FALSE)

# Stop it again
pa |> api_stop()
```

---

api_security_cors *Set up CORS for a path in your plumber2 API*

Description

This function adds Cross-Origin Resource Sharing (CORS) to a path in your API. The function can be called multiple times to set up CORS for multiple paths, potentially with different settings for each path. CORS is a complex specification and more can be read about it at the CORS plugin documentation.

Usage

```
api_security_cors(
  api,
  path = "/*",
  origin = "*",
  methods = c("get", "head", "put", "patch", "post", "delete"),
  allowed_headers = NULL,
  exposed_headers = NULL,
  allow_credentials = FALSE,
  max_age = NULL
)
```

Arguments

api	A plumber2 api object to add the plugin to
path	The path that the policy should apply to. routr path syntax applies, meaning that wilcards and path parameters are allowed.
origin	The origin allowed for the path. Can be one of: • A boolean. If TRUE then all origins are permitted and the preflight response will have the Access-Control-Allow-Origin header reflect the origin of the request. If FALSE then all origins are denied • The string "*" which will allow all origins and set Access-Control-Allow-Origin to *. This is different than setting it to TRUE because * instructs browsers that any origin is allowed and it may use this information when searching the cache • A character vector giving allowed origins. If the request origin matches any of these then the Access-Control-Allow-Origin header in the response will reflect the origin of the request • A function taking the request and returning TRUE if the origin is permitted and FALSE if it is not. If permitted the Access-Control-Allow-Origin header will reflect the request origin
methods	The HTTP methods allowed for the path
allowed_headers	A character vector of request headers allowed when making the request. If the request contains headers not permitted, then the response will be blocked by the browser. NULL will allow any header by reflecting the Access-Control-Request-Headers header value from the request into the Access-Control-Allow-Headers header in the response.
exposed_headers	A character vector of response headers that should be made available to the client upon a succesful request
allow_credentials	A boolean indicating whether credentials are allowed in the request. Credentials are cookies or HTTP authentication headers, which are normally stripped from fetch() requests by the browser. If this is TRUE then origin cannot be * according to the spec
max_age	The duration browsers are allowed to keep the preflight response in the cache

Value

This functions return the `api` object allowing for easy chaining with the pipe

Using annotation

To add CORS to a path you can add `@cors <origin>` to a handler annotation. `<origin>` must be one or more URLs or `*`, separated by comma (meaning it is not possible to provide a function using the annotation). This will add CORS to all endpoints described in the block. The annotation doesn't allow setting `allowed_headers`, `exposed_headers`, `allow_credentials` or `max_age` and the default values will be used.

```
#* A handler for /user/<username>
#*
#* @param username:string The name of the user to provide information on
#*
#* @get /user/<username>
#*
#* @response 200:{name:string, age:integer, hobbies:[string]} Important
#* information about the user such as their name, age, and hobbies
#*
#* @cors https://example.com, https://another-site.com
#*
function(username) {
  find_user_in_db(username)
}
```

See Also

Other security features: `api_security_headers()`, `api_security_resource_isolation()`

Examples

```
# Set up cors for your asset/ path for the https://examples.com origin

api() |>
  api_security_cors(
    path = "asset/*",
    origin = "https://examples.com"
  )
```

---

api_security_headers *Add various security related headers to your plumber2 API*

Description

This function adds the SecurityHeaders plugin to your plumber2 API. Please consult the documentation for the plugin for up-to-date information on its behaviour.

Usage

```
api_security_headers(
  api,
  content_security_policy = csp(default_src = "self", script_src = "self",
   script_src_attr = "none", style_src = c("self", "https:", "unsafe-inline"), img_src =
   c("self", "data:"), font_src = c("self", "https:", "data:"), object_src = "none",
    base_uri = "self", form_action = "self", frame_ancestors = "self",
    upgrade_insecure_requests = TRUE),
  content_security_policy_report_only = NULL,
  cross_origin_embedder_policy = NULL,
  cross_origin_opener_policy = "same-origin",
  cross_origin_resource_policy = "same-origin",
  origin_agent_cluster = TRUE,
  referrer_policy = "no-referrer",
 strict_transport_security = sts(max_age = 63072000, include_sub_domains = TRUE),
  x_content_type_options = TRUE,
  x_dns_prefetch_control = FALSE,
  x_download_options = TRUE,
  x_frame_options = "SAMEORIGIN",
  x_permitted_cross_domain_policies = "none",
  x_xss_protection = FALSE
)
```

Arguments

api	A plumber2 api object to add the plugin to
content_security_policy	Set the value of the `Content-Security-Policy` header. See `firesafety::csp()` for documentation of its values
content_security_policy_report_only	Set the value of the `Content-Security-Policy-Report-Only` header. See `firesafety::csp()` for documentation of its values
cross_origin_embedder_policy	Set the value of the `Cross-Origin-Embedder-Policy`. Possible values are `"unsafe-none"`, `"require-corp"`, and `"credentialless"`
cross_origin_opener_policy	Set the value of the `Cross-Origin-Opener-Policy`. Possible values are `"unsafe-none"`, `"same-origin-allow-popups"`, `"same-origin"`, and `"noopener-allow-popups"`
cross_origin_resource_policy	Set the value of the `Cross-Origin-Resource-Policy`. Possible values are `"same-site"`, `"same-origin"`, and `"cross-origin"`
origin_agent_cluster	Set the value of the `Origin-Agent-Cluster`. Possible values are `TRUE` and `FALSE`
referrer_policy	Set the value of the `Referrer-Policy`. Possible values are `"no-referrer"`, `"no-referrer-when-downgrade"`, `"origin"`, `"origin-when-cross-origin"`,

"same-origin", "strict-origin", "strict-origin-when-cross-origin", and "unsafe-url"

strict_transport_security
: Set the value of the Strict-Transport-Security header. See firesafety::sts() for documentation of its values

x_content_type_options
: Set the value of the X-Content-Type-Options. Possible values are TRUE and FALSE

x_dns_prefetch_control
: Set the value of the X-DNS-Prefetch-Control. Possible values are TRUE and FALSE

x_download_options
: Set the value of the X-Download-Options. Possible values are TRUE and FALSE

x_frame_options
: Set the value of the X-Frame-Options. Possible values are "DENY" and "SAMEORIGIN"

x_permitted_cross_domain_policies
: Set the value of the X-Permitted-Cross-Domain-Policies. Possible values are "none", "master-only", "by-content-type", "by-ftp-filename", "all", and "none-this-response"

x_xss_protection
: Set the value of the X-XSS-Protection. Possible values are TRUE and FALSE

Value

This functions return the api object allowing for easy chaining with the pipe

Using annotation

Security headers doesn't have a dedicated annotation tag, but you can set it up in a @plumber block

```
#* @plumber
function(api) {
  api |>
    api_security_headers()
}
```

See Also

Other security features: api_security_cors(), api_security_resource_isolation()

Examples

```
# Add default security headers to an API
api() |>
  api_security_headers()
```

`api_security_resource_isolation` *Set up resource isolation for a path*

Description

This function adds resource isolation to a path in your API. The function can be called multiple times to set up resource isolation for multiple paths, potentially with different settings for each path. You can read in depth about resource isolation at the ResourceIsolation plugin documentation.

Usage

```
api_security_resource_isolation(
  api,
  path = "/*",
  allowed_site = "same-site",
  forbidden_navigation = c("object", "embed"),
  allow_cors = TRUE
)
```

Arguments

`api`	A plumber2 api object to add the plugin to
`path`	The path that the policy should apply to. routr path syntax applies, meaning that wilcards and path parameters are allowed.
`allowed_site`	The allowance level to permit. Either `cross-site`, `same-site`, or `same-origin`.
`forbidden_navigation`	A vector of destinations not allowed for navigational requests. See the `Sec-Fetch-Dest` documentation for a description of possible values. The special value `"all"` is also permitted which is the equivalent of passing all values.
`allow_cors`	Should `Sec-Fetch-Mode: cors` requests be allowed

Value

This functions return the `api` object allowing for easy chaining with the pipe

Using annotation

To add resource isolation to a path you can add `@rip <allowed_site>` to a handler annotation. This will add resource isolation to all endpoints described in the block. The annotation doesn't allow setting `forbidden_navigation` or `allow_cors` and the default values will be used.

```
#* A handler for /user/<username>
#*
#* @param username:string The name of the user to provide information on
#*
```

```
#* @get /user/<username>
#*
#* @response 200:{name:string, age:integer, hobbies:[string]} Important
#* information about the user such as their name, age, and hobbies
#*
#* @rip same-origin
#*
function(username) {
  find_user_in_db(username)
}
```

See Also

Other security features: `api_security_cors()`, `api_security_headers()`

Examples

```
# Set up resource isolation for everything inside a user path
api() |>
  api_security_resource_isolation(
    path = "<user>/*"
  )
```

`api_session_cookie` *Turn on session cookie data storage for your API*

Description

If you need to keep data between requests, but don't want to store it server-side (see `api_datastore()`) you can instead pass it back and forth as an encrypted session cookie. This function sets it up on your api and after it's use you can now access and set session data in the request and response `$session` field. Be aware that session data is send back and forth with all requests and should thus be kept minimal to avoid congestion on your server.

Usage

```
api_session_cookie(
  api,
  key,
  name = "reqres",
  expires = NULL,
  max_age = NULL,
  path = NULL,
  secure = NULL,
  same_site = NULL
)
```

Arguments

api	A plumber2 api object to add the session cookie setup to
key	A 32-bit secret key as a hex encoded string or a raw vector to use for encrypting the session cookie. A valid key can be generated using `reqres::random_key()`. NEVER STORE THE KEY IN PLAIN TEXT. Optimally use the keyring package to store it
name	The name of the cookie
expires	A POSIXct object given the expiration time of the cookie
max_age	The number of seconds to elapse before the cookie expires
path	The URL path this cookie is related to
secure	Should the cookie only be send over https
same_site	Either `"Lax"`, `"Strict"`, or `"None"` indicating how the cookie can be send during cross-site requests. If this is set to `"None"` then `secure` *must* also be set to `TRUE`

Value

These functions return the `api` object allowing for easy chaining with the pipe

Using annotation

Session cookie setup doesn't have a dedicated annotation tag, but you can set it up in a `@plumber` block

```
#* @plumber
function(api) {
  api |>
    api_session_cookie(keyring::key_get("my_secret_plumber_key"))
}
```

Examples

```
key <- reqres::random_key()

api() |>
  api_session_cookie(key, secure = TRUE) |>
  api_get("/", function(request) {
    if (isTRUE(request$session$foo)) {
      msg <- "You've been here before"
    } else {
      msg <- "You must be new here"
      request$session$foo <- TRUE
    }
    list(
      msg = msg
    )
  })
```

`api_shiny` *Serve a Shiny app from a plumber2 api*

Description

You can serve one or more shiny apps as part of a plumber2 api. The shiny app launches in a background process and the api will work as a reverse proxy to forward requests to `path` to the process and relay the response to the client. The shiny app is started along with the api and shut down once the api is stopped. This functionality requires the shiny and callr packages to be installed. Be aware that all requests to subpaths of `path` will be forwarded to the shiny process, and thus not end up in your normal route

Usage

```
api_shiny(api, path, app, except = NULL)
```

Arguments

`api`	A plumber2 api to add the shiny app to
`path`	The path to serve the shiny app from
`app`	A shiny app object
`except`	Subpaths to `path` that should not be forwarded to the shiny app. Be sure it doesn't contains paths that the shiny app needs

Value

This functions return the `api` object allowing for easy chaining with the pipe

Using annotation

A shiny app can be served using an annotated route file by using the `@shiny` tag and proceeding the annotation block with the shiny app object

```
#* @shiny /my_app/
shiny::shinyAppDir("./shiny")
```

Examples

```
blank_shiny <- shiny::shinyApp(
  ui = shiny::fluidPage(),
  server = shiny::shinyServer(function(...) {})
)

api() |>
  api_shiny("my_app/", blank_shiny)
```

async_evaluators *Async evaluators provided by plumber*

Description

These functions support async request handling. You can register your own as well using `register_async()`.

Usage

```
mirai_async(...)
```

Arguments

`...`	Further argument passed on to the internal async function. See Details for information on which function handles the formatting internally in each async evaluator

Value

A function taking `expr` and `envir`. The former is the expression to evaluate and the latter is an environment with additional variables that should be made available during evaluation

Provided evaluators

- `mirai_async()` uses `mirai::mirai()`. It is registered as `"mirai"`. Be aware that for this evaluator to be performant you should start up multiple persistent background processes. See `mirai::daemons()`.

Examples

```
# Use the default mirai backend by setting `async = TRUE` with a handler

pa <- api() |>
  api_get("/hello/<name:string>", function(name) {
    list(
      msg = paste0("Hello ", name, "!")
    )
  }, async = TRUE)
```

create_server_yml *Create a _server.yml file to describe your API*

Description

While you can manually create a plumber2 API by calling `api()`, you will often need to deploy the api somewhere else. To facilitate this you can create a `_server.yml` that encapsulates all of your settings and plumber files. If you call `api()` with a path to such a file the API will be constructed according to its content.

Usage

```
create_server_yml(..., path = ".", constructor = NULL, freeze_opt = TRUE)
```

Arguments

`...`	path to files and/or directories that contain annotated plumber files to be used by your API
`path`	The folder to place the generated `_server.yml` file in
`constructor`	The path to a file that creates a plumber2 API object. Can be omitted in which case an API object will be created for you
`freeze_opt`	Logical specifying whether any options you currently have locally (either as environment variables or R options) should be written to the `_server.yml` file. Shared secret will never be written to the file and you must find a different way to move that to your deployment server.

Examples

```
create_server_yml(
  "path/to/a/plumber/file.R"
)
```

get_opts *Retrieve options for creating a plumber2 api*

Description

You can provide options for your plumber2 api which will be picked up when you create the API with `api()`. Options can be set either through the internal `options()` functionality, or by setting environment variables. In the former case, the name of the option must be prefixed with `"plumber2."`, in the latter case the variable name must be in upper case and prefixed with `"PLUMBER2_"`. If the option is stored as an environment variable then the value is cast to the type giving in `default`. See the docs for `api()` for the default values of the different options.

Usage

```
get_opts(x, default = NULL)

all_opts()
```

Arguments

x	The name of the option
default	The default value, if x is not set

Value

For get_opts The value of x, if any, or default. For all_opts() a named list of all the options that are set

Examples

```
# Using `options()`
old_opts <- options(plumber2.port = 9889L)
get_opts("port")
options(old_opts)

# Using environment variables
old_env <- Sys.getenv("PLUMBER2_PORT")
Sys.setenv(PLUMBER2_PORT = 9889)

## If no default is provided the return value is a string
get_opts("port")

## Provide a default to hint at the options type
get_opts("port", 8080L)

Sys.setenv(PLUMBER2_PORT = old_env)
```

Next *Router control flow*

Description

In plumber2 your API can have multiple middleware that a request passes through. At any point can you short-circuit the remaining middleware by returning Break, which instructs plumber2 to return the response as is. Returning Next indicates the opposite, ie that the request should be allowed to pass on to the next middleware in the chain. A handler function that doesn't return either of these are assumed to return a value that should be set to the response body and implicitely continue to the next middleware.

Usage

```
Next

Break

should_break(x)
```

Arguments

x	An object to test

Value

A boolean value

Examples

```
# should_break() only returns TRUE with Break

should_break(10)

should_break(FALSE)

should_break(Next)

should_break(Break)
```

---

openapi *Construct OpenAPI specifications*

Description

These helper functions aid in constructing OpenAPI compliant specifications for your API. The return simple lists and you may thus forego these helpers and instead construct it all manually (or import it from a json or yaml file). The purpose of these helpers is mainly in basic input checking and for documenting the structure. Read more about the spec at https://spec.openapis.org/oas/v3.0.0.html

Usage

```
openapi(
  openapi = "3.0.0",
  info = openapi_info(),
  paths = list(),
  tags = list()
)

openapi_info(
```

```
  title = character(),
  description = character(),
  terms_of_service = character(),
  contact = openapi_contact(),
  license = openapi_license(),
  version = character()
)

openapi_contact(name = character(), url = character(), email = character())

openapi_license(name = character(), url = character())

openapi_path(
  summary = character(),
  description = character(),
  get = openapi_operation(),
  put = openapi_operation(),
  post = openapi_operation(),
  delete = openapi_operation(),
  options = openapi_operation(),
  head = openapi_operation(),
  patch = openapi_operation(),
  trace = openapi_operation(),
  parameters = list()
)

openapi_operation(
  summary = character(),
  description = character(),
  operation_id = character(),
  parameters = list(),
  request_body = openapi_request_body(),
  responses = list(),
  tags = character()
)

openapi_parameter(
  name = character(),
  location = c("path", "query", "header", "cookie"),
  description = character(),
  required = logical(),
  schema = openapi_schema(),
  content = openapi_content(),
  ...
)

openapi_header(description = character(), schema = openapi_schema())
```

```
openapi_schema(x, default = NULL, min = NULL, max = NULL, ..., required = NULL)

openapi_content(...)

openapi_request_body(
  description = character(),
  content = openapi_content(),
  required = logical()
)

openapi_response(
  description = character(),
  content = openapi_content(),
  headers = list()
)

openapi_tag(name = character(), description = character())
```

Arguments

openapi	The OpenAPI version the spec adheres to. The helpers assume 3.0.0 so this is also the default value
info	A list as constructed by openapi_info()
paths	A named list. The names correspond to endpoints and the elements are lists as constructed by openapi_path()
tags	For openapi() a list with elements corresponding to the value constructed by openapi_tag(). For openapi_operation() a character vector or a list of strings
title	A string giving the title of the API
description	A longer description of the respective element. May use markdown
terms_of_service	A URL to the terms of service for the API
contact	A list as constructed by openapi_contact()
license	A list as constructed by openapi_license()
version	A string giving the version of the API
name	The name of the contact, license, parameter, or tag
url	The URL pointing to the contact or license information
email	An email address for the contact
summary	A one-sentence summary of the path or operation
get, put, post, delete, options, head, patch, trace	A list describing the specific HTTP method when requested for the path, as constructed by openapi_operation()
parameters	A list of parameters that apply to the path and/or operation. If this is given in openapi_path() it is inherited by all its operations.

operation_id	A unique string that identifies this operation in the API
request_body	A list as constructed by openapi_request_body()
responses	A named list with the name corresponding to the response code and the elements being lists as constructed by openapi_response()
location	Where this parameter is coming from. Either "path", "query", "header", or "cookie".
required	For openapi_parameter a boolean indicating if this is a required parameter ("path" parameters are always required). For openapi_schema() a character vector naming the required properties of an object.
schema	A description of the data as constructed by openapi_schema
content	A list as constructed by openapi_content().
...	Further named arguments to be added to the element. For openapi_content() named elements as constructed by openapi_schema()
x	An R object corresponding to the type of the schema. Supported types are: • integer: Will signal type: integer • numeric: Will signal type: number • character: Will signal type: string • factor: Will signal type: string and enum set the factor levels • raw: Will signal type:string; format: binary • Date: Will signal type:string; format: date • POSIXt: Will signal type:string; format: date-time • list: If unnamed it must be a one-length list and will signal type: array and items set to the schema of its element. If named it will signal type: object and properties set to the schema of each element. • AsIs: Will signal a type equivalent to the value of the input (must be a string)
default	A default value for the parameter. Must be reconsilable with the type of x
min, max	Bounds for the value of the parameter
headers	A named list with names corresponding to headers and elements as constructed by openapi_header()

Value

A list

Examples

```
# Create docs for an API with a single endpoint
doc <- openapi(
  info = openapi_info(
    title = "My awesome api",
    version = "1.0.0"
  ),
  paths = list(
    "/hello/{name}" = openapi_path(
```

```
        get = openapi_operation(
          summary = "Get a greeting",
          parameters = list(
            openapi_parameter(
              name = "name",
              location = "path",
              description = "Your name",
              schema = openapi_schema(character())
            )
          ),
          responses = list(
            "200" = openapi_response(
              description = "a kind message",
              content = openapi_content(
                "text/plain" = openapi_schema(character())
              )
            )
          )
        )
      )
    )
  )

  # Add it to an api
  api() |>
    api_doc_add(doc)
```

`parsers` *Parser functions provided by plumber2*

Description

These functions cover a large area of potential request body formats. They are all registered to their standard mime types but users may want to use them to register them to alternative types if they know it makes sense.

Usage

```
parse_csv(...)

parse_octet()

parse_rds(...)

parse_feather(...)

parse_parquet(...)
```

```
parse_text(multiple = FALSE)

parse_tsv(...)

parse_yaml(...)

parse_geojson(...)

parse_multipart(parsers = get_parsers())
```

Arguments

...	Further argument passed on to the internal parsing function. See Details for information on which function handles the parsing internally in each parser
multiple	logical: should the conversion be to a single character string or multiple individual characters?
parsers	A list of parsers to use for parsing the parts of the body

Value

A function accepting a raw vector along with a `directives` argument that provides further directives from the `Content-Type` to be passed along

Provided parsers

- `parse_csv()` uses `readr::read_csv()` for parsing. It is registered as `"csv"` for the mime types `application/csv`, `application/x-csv`, `text/csv`, and `text/x-csv`
- `parse_multipart` uses `webutils::parse_multipart()` for the initial parsing. It then goes through each part and tries to find a parser that matches the content type (either given directly or guessed from the file extension provided). If a parser is not found it leaves the value as a raw vector. It is registered as `"multi"` for the mime type `multipart/*`
- `parse_octet()` passes the raw data through unchanged. It is registered as `"octet"` for the mime type `application/octet-stream`
- `parse_rds()` uses `unserialize()` for parsing. It is registered as `"rds"` for the mime type `application/rds`
- `parse_feather()` uses `arrow::read_feather()` for parsing. It is registered as `"feather"` for the mime types `application/vnd.apache.arrow.file` and `application/feather`
- `parse_parquet()` uses `arrow::read_parquet()` for parsing. It is registered as `"parquet"` for the mime type `application/vnd.apache.parquet`
- `parse_text()` uses `rawToChar()` for parsing. It is registered as `"text"` for the mime types `text/plain` and `text/*`
- `parse_tsv()` uses `readr::read_tsv()` for parsing. It is registered as `"tsv"` for the mime types `application/tab-separated-values` and `text/tab-separated-values`
- `parse_yaml()` uses `yaml::yaml.load()` for parsing. It is registered as `"yaml"` for the mime types `text/vnd.yaml`, `application/yaml`, `application/x-yaml`, `text/yaml`, and `text/x-yaml`

- parse_geojson() uses geojsonsf::geojson_sf() for parsing. It is registered as "geojson" for the mime types application/geo+json and application/vdn.geo+json

Additional registered parsers:

- reqres::parse_json() is registered as "json" for the mime types application/json and text/json
- reqres::parse_queryform() is registered as "form" for the mime type application/x-www-form-urlencoded
- reqres::parse_xml() is registered as "xml" for the mime types application/xml and text/xml
- reqres::parse_html() is registered as "html" for the mime type text/html

See Also

register_parser()

Examples

```
# You can use parsers directly when adding handlers
pa <- api() |>
  api_post("/hello/<name:string>", function(name, body) {
    list(
      msg = paste0("Hello ", name, "!")
    )
  }, parsers = list("text/csv" = parse_csv()))
```

Plumber2 *The Plumber2 Class*

Description

This class encapsulates all of the logic of a plumber2 api, and is what gets passed around in the functional api of plumber2. The Plumber2 class is a subclass of the fiery::Fire class. Please consult the documentation for this for additional information on what this type of server is capable of. Note that the Plumber2 objects are reference objects, meaning that any change to it will change all instances of the object.

Initialization:

A new Plumber2-object is initialized using the new() method on the generator:

```
api <- Plumber2$new()
```

However, most users will use the functional api of the package and thus construct one using api()

Copying:

As Plumber2 objects are using reference semantics new copies of an api cannot be made simply be assigning it to a new variable. If a true copy of a Plumber2 object is desired, use the clone() method.

Super class

`fiery::Fire` -> `Plumber2`

Active bindings

`request_router` The router handling requests

`header_router` The router handling partial requests (the request will pass through this router prior to reading in the body)

`doc_type` The type of API documentation to generate. Can be either `"rapidoc"` (the default), `"redoc"`, `"swagger"`, or `NULL` (equating to not generating API docs)

`doc_path` The URL path to serve the api documentation from

Methods

Public methods:

- `Plumber2$new()`
- `Plumber2$format()`
- `Plumber2$ignite()`
- `Plumber2$add_route()`
- `Plumber2$request_handler()`
- `Plumber2$message_handler()`
- `Plumber2$redirect()`
- `Plumber2$parse_file()`
- `Plumber2$add_api_doc()`
- `Plumber2$add_shiny()`
- `Plumber2$forward()`
- `Plumber2$clone()`

Method `new()`**:** Create a new `Plumber2` api

Usage:

```
Plumber2$new(
  host = get_opts("host", "127.0.0.1"),
  port = get_opts("port", 8080),
  doc_type = get_opts("docType", "rapidoc"),
  doc_path = get_opts("docPath", "__docs__"),
  reject_missing_methods = get_opts("rejectMissingMethods", FALSE),
  ignore_trailing_slash = get_opts("ignoreTrailingSlash", TRUE),
  max_request_size = get_opts("maxRequestSize"),
  shared_secret = get_opts("sharedSecret"),
  compression_limit = get_opts("compressionLimit", 1000),
  default_async = get_opts("async", "mirai"),
  env = caller_env()
)
```

Arguments:

`host` A string overriding the default host

`port` An port number overriding the default port

`doc_type` The type of API documentation to generate. Can be either `"rapidoc"` (the default), `"redoc"`, `"swagger"`, or `NULL` (equating to not generating API docs)

`doc_path` The URL path to serve the api documentation from

`reject_missing_methods` Should requests to paths that doesn't have a handler for the specific method automatically be rejected with a 405 Method Not Allowed response with the correct Allow header informing the client of the implemented methods. Assigning a handler to `"any"` for the same path at a later point will overwrite this functionality. Be aware that setting this to `TRUE` will prevent the request from falling through to other routes that might have a matching method and path. This setting anly affects handlers on the request router.

`ignore_trailing_slash` Logical. Should the trailing slash of a path be ignored when adding handlers and handling requests. Setting this will not change the request or the path associated with but just ensure that both `path/to/resource` and `path/to/resource/` ends up in the same handler. This setting will only affect routes that are created automatically.

`max_request_size` Sets a maximum size of request bodies. Setting this will add a handler to the header router that automatically rejects requests based on their `Content-Length` header

`shared_secret` Assigns a shared secret to the api. Setting this will add a handler to the header router that automatically rejects requests if their `Plumber-Shared-Secret` header doesn't contain the same value. Be aware that this type of authentication is very weak. Never put the shared secret in plain text but rely on e.g. the keyring package for storage. Even so, if requests are send over HTTP (not HTTPS) then anyone can read the secret and use it

`compression_limit` The size threshold in bytes for trying to compress the response body (it is still dependant on content negotiation)

`default_async` The default evaluator to use for async request handling

`env` An environment that will be used as the default execution environment for the API

Returns: A `Plumber2` object

Method `format()`**:** Human readable description of the api object

Usage:

```
Plumber2$format(...)
```

Arguments:

`...` ignored

Returns: A character vector

Method `ignite()`**:** Begin running the server. Will trigger the `start` event

Usage:

```
Plumber2$ignite(
  block = FALSE,
  showcase = is_interactive(),
  ...,
  silent = FALSE
)
```

Arguments:

`block` Should the console be blocked while running (alternative is to run in the background)

`showcase` Should the default browser open up at the server address. If `TRUE` then a browser opens at the root of the api, unless the api contains OpenAPI documentation in which case it will open at that location. If a string the string is used as a path to add to the root before opening.

`...` Arguments passed on to the `start` handler

`silent` Should startup messaging by silenced

Method `add_route()`**:** Add a new route to either the request or header router

Usage:

```
Plumber2$add_route(name, route = NULL, header = FALSE, after = NULL, root = "")
```

Arguments:

`name` The name of the route to add. If a route is already present with this name then the provided route (if any) is merged into it

`route` The route to add. If `NULL` a new empty route will be created

`header` Logical. Should the route be added to the header router?

`after` The location to place the new route on the stack. `NULL` will place it at the end. Will not have an effect if a route with the given name already exists.

`root` The root path to serve this route from.

Method `request_handler()`**:** Add a handler to a request. See api_request_handlers for detailed information

Usage:

```
Plumber2$request_handler(
  method,
  path,
  handler,
  serializers,
  parsers = NULL,
  use_strict_serializer = FALSE,
  download = FALSE,
  async = FALSE,
  then = NULL,
  doc = NULL,
  route = NULL,
  header = FALSE
)
```

Arguments:

`method` The HTTP method to attach the handler to

`path` A string giving the path the handler responds to.

`handler` A handler function to call when a request is matched to the path

`serializers` A named list of serializers that can be used to format the response before sending it back to the client. Which one is selected is based on the request `Accept` header

`parsers` A named list of parsers that can be used to parse the request body before passing it in as the body argument. Which one is selected is based on the request `Content-Type` header

use_strict_serializer By default, if a serializer that respects the requests Accept header cannot be found, then the first of the provided ones are used. Setting this to TRUE will instead send back a 406 Not Acceptable response

download Should the response mark itself for download instead of being shown inline? Setting this to TRUE will set the Content-Disposition header in the response to attachment. Setting it to a string is equivalent to setting it to TRUE but will in addition also set the default filename of the download to the string value

async If FALSE create a regular handler. If TRUE, use the default async evaluator to create an async handler. If a string, the async evaluator registered to that name is used. If a function is provided then this is used as the async evaluator

then A function to call at the completion of an async handler

doc OpenAPI documentation for the handler. Will be added to the paths$<handler_path>$<handler_method> portion of the API.

route The route this handler should be added to. Defaults to the last route in the stack. If the route does not exist it will be created as the last route in the stack.

header Logical. Should the handler be added to the header router

Method message_handler(): Add a handler to a WebSocket message. See api_message for detailed information

Usage:

```
Plumber2$message_handler(handler, async = FALSE, then = NULL)
```

Arguments:

handler A function conforming to the specifications laid out in api_message()

async If FALSE create a regular handler. If TRUE, use the default async evaluator to create an async handler. If a string, the async evaluator registered to that name is used. If a function is provided then this is used as the async evaluator

then A function to call at the completion of an async handler

Method redirect(): Add a redirect to the header router. Depending on the value of permanent it will respond with a 307 Temporary Redirect or 308 Permanent Redirect. from and to can contain path parameters and wildcards which will be matched between the two to construct the correct redirect path.

Usage:

```
Plumber2$redirect(method, from, to, permanent = TRUE)
```

Arguments:

method The HTTP method the redirect should respond to

from The path the redirect should respond to

to The path/URL to redirect the incoming request towards. After resolving any path parameters and wildcards it will be used in the Location header

permanent Logical. Is the redirect considered permanent or temporary? Determines the type of redirct status code to use

Method parse_file(): Parses a plumber file and updates the app according to it

Usage:

```
Plumber2$parse_file(file, env = NULL)
```

Arguments:

`file` The path to a file to parse

`env` The parent environment to the environment the file should be evaluated in. If `NULL` the environment provided at construction will be used

Method `add_api_doc()`**:** Add a (partial) OpenAPI spec to the api docs

Usage:

```
Plumber2$add_api_doc(doc, overwrite = FALSE, subset = NULL)
```

Arguments:

`doc` A list with the OpenAPI documentation

`overwrite` Logical. Should already existing documentation be removed or should it be merged together with `doc`

`subset` A character vector giving the path to the subset of the docs to assign `doc` to

Method `add_shiny()`**:** Add a shiny app to an api. See `api_shiny()` for detailed information

Usage:

```
Plumber2$add_shiny(path, app, except = NULL)
```

Arguments:

`path` The path to serve the app from

`app` A shiny app object

`except` Subpaths to `path` that should not be forwarded to the shiny app. Be sure it doesn't contains paths that the shiny app needs

Method `forward()`**:** Add a reverse proxy from a path to a given URL. See `api_forward()` for more details

Usage:

```
Plumber2$forward(path, url, except = NULL)
```

Arguments:

`path` The root to forward from

`url` The url to forward to

`except` Subpaths to `path` that should be exempt from forwarding

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

Usage:

```
Plumber2$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

`register_async` *Register an async evaluator*

Description

plumber supports async request handling in two ways. Either manual by returning a promise from the handler, or automatic through the `@async` tag / `async` argument in the handler functions. The default evaluator is controlled by the `plumber2.async` option or the `PLUMBER2_ASYNC` environment variable.

Usage

```
register_async(name, fun, dependency = NULL)

show_registered_async()

get_async(name = NULL, ...)
```

Arguments

`name`	The name of the evaluator
`fun`	A function that, upon calling it returns an evaluator taking an `expr` and `envir` argument. See the async evaluator functions for examples
`dependency`	Package dependencies for the evaluator.
`...`	Arguments passed on to the async function creator

Examples

```
# Register an async evaluator based on future (the provided mirai backend is
# superior in every way so this is for illustrative purpose)
future_async <- function(...) {
  function(expr, envir) {
    promises::future_promise(
      expr = expr,
      envir = envir,
      substitute = FALSE,
      ...
    )
  }
}
register_async("future", future_async, c("promises", "future"))
```

`register_parser` *Register or fetch a parser*

Description

plumber2 comes with many parsers that should cover almost all standard use cases. Still you might want to provide some of your own, which this function facilitates.

Usage

```
register_parser(name, fun, mime_types, default = TRUE)

show_registered_parsers()

get_parsers(parsers = NULL)
```

Arguments

`name`	The name to register the parser function to. If already present the current parser will be overwritten by the one provided by you
`fun`	A function that, when called, returns a binary function that can parse a request body. The first argument takes a raw vector with the binary encoding of the request body, the second argument takes any additional directives given by the requests `Content-Type` header
`mime_types`	One or more mime types that this parser can handle. The mime types are allowed to contain wildcards, e.g. `"text/*"`
`default`	Should this parser be part of the default set of parsers
`parsers`	Parsers to collect. This can either be a character vector of names of registered parsers or a list. If it is a list then the following expectations apply: • Any unnamed elements containing a character vector will be considered as names of registered parsers constructed with default values. The special value `"..."` will fetch all the parsers that are otherwise not specified in the call • Any element containing a function are considered as a provided parser and the element must be named by the mime type the parser understands (wildcards allowed) • Any remaining named elements will be considered names of registered parsers that should be constructed with the arguments given in the element

Details

If you want to register your own parser, then the function you register must be a factory function, i.e. a function returning a function. The returned function must accept two arguments, the first being a raw vector corresponding to the request body, the second being the parsed directives from the request `Content-Type` header. All arguments to the factory function should be optional.

Value

For `get_parsers` a named list of parser functions named by their mime types. The order given in `parsers` is preserved.

See Also

parsers

`register_serializer()`

Examples

```
# Register a parser that splits at a character and converts to number
register_parser("comma", function(delim = ",") {
  function(raw, directive) {
    as.numeric(strsplit(rawToChar(raw), delim)[[1]])
  }
}, mime_types = "text/plain", default = FALSE)
```

`register_serializer` *Register or fetch a serializer*

Description

plumber2 comes with many serializers that should cover almost all standard use cases. Still you might want to provide some of your own, which this function facilitates.

Usage

```
register_serializer(name, fun, mime_type, default = TRUE)

show_registered_serializers()

get_serializers(serializers = NULL)
```

Arguments

`name`	The name to register the serializer function to. If already present the current serializer will be overwritten by the one provided by you
`fun`	A function that, when called, returns a unary function that can serialize a response body to the mime type defined in `mime_type`
`mime_type`	The format this serializer creates. You should take care to ensure that the value provided is a standard mime type for the format
`default`	Should this serializer be part of the default set of serializers
`serializers`	Serializers to collect. This can either be a character vector of names of registered serializers or a list. If it is a list then the following expectations apply:

- Any unnamed elements containing a character vector will be considered as names of registered serializers constructed with default values. The special value "..." will fetch all the serializers that are otherwise not specified in the call.
- Any element containing a function are considered as a provided serializer and the element must be named by the mime type the serializer understands
- Any remaining named elements will be considered names of registered serializers that should be constructed with the arguments given in the element

Details

If you want to register your own serializer, then the function you register must be a factory function, i.e. a function returning a function. The returned function must accept a single argument which is the response body. All arguments to the factory function should be optional.

Value

For `get_serializers` a named list of serializer functions named by their mime type. The order given in `serializers` is preserved.

Note

Using the `...` will provide remaining graphics serializers if a graphics serializer is explicitely requested elsewhere. Otherwise it will provide the remaining non-graphics serializers. A warning is thrown if a mix of graphics and non-graphics serializers are requested.

See Also

serializers

`register_serializer()`

Examples

```
# Add a serializer that deparses the value
register_serializer("deparse", function(...) {
  function(x) {
    deparse(x, ...)
  }
}, mime_type = "text/plain")
```

serializers *Serializer functions provided by plumber2*

Description

These functions cover a large area of potential response body formats. They are all registered to their standard mime type but users may want to use them to register them to alternative types if they know it makes sense.

Usage

```
format_csv(...)

format_tsv(...)

format_rds(version = "3", ascii = FALSE, ...)

format_geojson(...)

format_feather(...)

format_parquet(...)

format_yaml(...)

format_htmlwidget(...)

format_format(..., sep = "\n")

format_print(..., sep = "\n")

format_cat(..., sep = "\n")

format_unboxed(...)

format_png(...)

format_jpeg(...)

format_tiff(...)

format_svg(...)

format_bmp(...)

format_pdf(...)
```

Arguments

...	Further argument passed on to the internal formatting function. See Details for information on which function handles the formatting internally in each serializer
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.
sep	The separator between multiple elements

Value

A function accepting the response body

Provided serializers

- format_csv() uses readr::format_csv() for formatting. It is registered as "csv" to the mime type text/csv
- format_tsv() uses readr::format_tsv() for formatting. It is registered as "tsv" to the mime type text/tsv
- format_rds() uses serialize() for formatting. It is registered as "rds" to the mime type application/rds
- format_geojson() uses geojsonsf::sfc_geojson() or geojsonsf::sf_geojson() for formatting depending on the class of the response body. It is registered as "geojson" to the mime type application/geo+json
- format_feather() uses arrow::write_feather() for formatting. It is registered as "feather" to the mime type application/vnd.apache.arrow.file
- format_parquet() uses nanoparquet::write_parquet() for formatting. It is registered as "parquet" to the mime type application/vnd.apache.parquet
- format_yaml() uses yaml::as.yaml() for formatting. It is registered as "yaml" to the mime type text/yaml
- format_htmlwidget() uses htmlwidgets::saveWidget() for formatting. It is registered as "htmlwidget" to the mime type text/html
- format_format() uses format() for formatting. It is registered as "format" to the mime type text/plain
- format_print() uses print() for formatting. It is registered as "print" to the mime type text/plain
- format_cat() uses cat() for formatting. It is registered as "cat" to the mime type text/plain
- format_unboxed() uses reqres::format_json() with auto_unbox = TRUE for formatting. It is registered as "unboxedJSON" to the mime type application/json

Additional registered serializers:

- reqres::format_json() is registered as "json" to the mime type application/json
- reqres::format_html() is registered as "html" to the mime type text/html
- reqres::format_xml() is registered as "xml" to the mime type text/xml
- reqres::format_plain() is registered as "text" to the mime type text/plain

Provided graphics serializers

Serializing graphic output is special because it requires operations before and after the handler is executed. Further, handlers creating graphics are expected to do so through side-effects (i.e. call to graphics rendering) or by returning a ggplot2 object. If you want to create your own graphics serializer you should use `device_formatter()` for constructing it.

- `format_png()` uses `ragg::agg_png()` for rendering. It is registered as `"png"` to the mime type `image/png`
- `format_jpeg()` uses `ragg::agg_jpeg()` for rendering. It is registered as `"jpeg"` to the mime type `image/jpeg`
- `format_tiff()` uses `ragg::agg_tiff()` for rendering. It is registered as `"tiff"` to the mime type `image/tiff`
- `format_svg()` uses `svglite::svglite()` for rendering. It is registered as `"svg"` to the mime type `image/svg+xml`
- `format_bmp()` uses `grDevices::bmp()` for rendering. It is registered as `"bmp"` to the mime type `image/bmp`
- `format_pdf()` uses `grDevices::pdf()` for rendering. It is registered as `"pdf"` to the mime type `application/pdf`

See Also

`register_serializer()`

Examples

```
# You can use serializers directly when adding handlers
pa <- api() |>
  api_get("/hello/<name:string>", function(name) {
    list(
      msg = paste0("Hello ", name, "!")
    )
  }, serializers = list("application/json" = format_unboxed()))
```

Index