Package 'fiery'

January 12, 2026

Type Package

Title A Lightweight and Flexible Web Framework

Version 1.5.0

Description A very flexible framework for building server side logic in R. The framework is unopinionated when it comes to how HTTP requests and WebSocket messages are handled and supports all levels of app complexity; from serving static content to full-blown dynamic web-apps. Fiery does not hold your hand as much as e.g. the shiny package does, but instead sets you free to create your web app the way you want.

License MIT + file LICENSE

URL https://fiery.data-imaginist.com, https://github.com/thomasp85/fiery

BugReports https://github.com/thomasp85/fiery/issues

Imports cli, fs, glue, httpuv, later, lifecycle, otel, promises, R6, reqres (>= 1.2.0), rlang (>= 1.1.0), sodium, stats, stringi, utils, yaml

Suggests callr, covr, future, knitr, logger, rmarkdown, testthat (>= 3.3.0), withr

VignetteBuilder knitr

Encoding UTF-8

RoxygenNote 7.3.3

Collate 'loggers.R' 'aaa.R' 'HandlerStack.R' 'Fire.R' 'PromiseStack.R' 'additional_type_checks.R' 'fake_request.R' 'fiery-package.R' 'import-standalone-obj-type.R' 'import-standalone-types-check.R' 'otel.R' 'request_store.R' 'zzz.R'

Config/testthat/edition 3

NeedsCompilation no

Author Thomas Lin Pedersen [cre, aut] (ORCID: <https://orcid.org/0000-0002-5147-4711>)

Maintainer Thomas Lin Pedersen <thomasp85@gmail.com>

Repository CRAN

Date/Publication 2026-01-12 09:00:02 UTC

Contents

`Fire` *Generate a New App Object*

Description

The Fire generator creates a new `Fire`-object, which is the class containing all the app logic. The class is based on the R6 OO-system and is thus reference-based with methods and data attached to each object, in contrast to the more well known S3 and S4 systems. A `fiery` server is event driven, which means that it is build up and manipulated by adding event handlers and triggering events. To learn more about the `fiery` event model, read the event vignette. `fiery` servers can be modified directly or by attaching plugins. As with events, plugins has its own vignette.

Initialization:

A new 'Fire'-object is initialized using the `new()` method on the generator:

```
app <- Fire$new(host = '127.0.0.1', port = 8080L)
```

Copying:

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

Active bindings

`host` A string giving a valid IPv4 address owned by the server, or `'0.0.0.0'` to listen on all addresses. The default is `'127.0.0.1'`

`port` An integer giving the port number the server should listen on (defaults to `8080L`)

`refresh_rate` **[Deprecated]** The interval in seconds between run cycles when running a blocking server (defaults to `0.001`)

`refresh_rate_nb` The interval in seconds between run cycles when running a non-blocking server (defaults to `1`)

`trigger_dir` A valid folder where trigger files can be put when running a blocking server (defaults to `NULL`). See the *The event cycle in fiery* vignette for more information.

`plugins` A named list of the already attached plugins. **Read Only** - can only be modified using the `attach()` method.

`data_store` Access the environment that holds the global data store

`root` The location of the app. Setting this will remove the root value from requests (or decline them with `400` if the request does not match the root). E.g. the path of a request will be changed from `/demo/test` to `/test` if `root == '/demo'`

`access_log_format` A glue string defining how requests will be logged. For standard formats see common_log_format and combined_log_format. Defaults to the *Common Log Format*

`key` The encryption key to use for request/response encryption

`session_cookie_settings` Get or set the session cookie settings

`trust` A logical indicating whether incoming requests are trusted.

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

`query_delim` The delimeter used to split array-type query arguments when parsing the query string

Methods

Public methods:

- `Fire$new()`
- `Fire$format()`
- `Fire$ignite()`
- `Fire$start()`
- `Fire$reignite()`
- `Fire$resume()`
- `Fire$extinguish()`
- `Fire$stop()`
- `Fire$on()`
- `Fire$off()`
- `Fire$trigger()`
- `Fire$send()`
- `Fire$close_ws_con()`
- `Fire$serve_static()`
- `Fire$exclude_static()`
- `Fire$attach()`
- `Fire$has_plugin()`
- `Fire$header()`
- `Fire$set_data()`
- `Fire$get_data()`
- `Fire$remove_data()`
- `Fire$time()`
- `Fire$remove_time()`
- `Fire$delay()`
- `Fire$remove_delay()`
- `Fire$async()`

- Fire$remove_async()
- Fire$set_client_id_converter()
- Fire$set_logger()
- Fire$log()
- Fire$is_running()
- Fire$safe_call()
- Fire$test_request()
- Fire$test_header()
- Fire$test_message()
- Fire$test_websocket()
- Fire$clone()

Method `new()`**:** Create a new `Fire` app

Usage:

```
Fire$new(host = "127.0.0.1", port = 8080)
```

Arguments:

`host` A string overriding the default host

`port` An port number overriding the default port

Returns: A `Fire` object

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

Usage:

```
Fire$format(...)
```

Arguments:

`...` ignored

Returns: A character vector

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

Usage:

```
Fire$ignite(block = TRUE, showcase = FALSE, ..., 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 app. 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 `start()`**:** Synonymous method to `ignite()`

Usage:

```
Fire$start(...)
```

Arguments:

`...` passed on to `ignite()`

Method `reignite()`**:** Resume a session. This is equivalent to `ignite()` but will also trigger the `resume` event

Usage:

```
Fire$reignite(...)
```

Arguments:

`...` passed on to `ignite()`

Method `resume()`**:** Synonymous method to `reignite()`

Usage:

```
Fire$resume(...)
```

Arguments:

`...` passed on to `ignite()`

Method `extinguish()`**:** Stop the server. Will trigger the `end` event

Usage:

```
Fire$extinguish()
```

Method `stop()`**:** Synonymous method to `extinguish()`

Usage:

```
Fire$stop()
```

Method `on()`**:** Add a handler to an event. See the *The event cycle in fiery* vignette for more information.

Usage:

```
Fire$on(event, handler, pos = NULL, id = NULL)
```

Arguments:

`event` The name of the event that should trigger the handler

`handler` The handler function that should be triggered

`pos` The position in the handler stack to place it at. `NULL` will place it at the end.

`id` An optional id to use to identify the handler

Returns: A unique string identifying the handler (either `id` or generated for you)

Method `off()`**:** Remove an event handler from the app.

Usage:

```
Fire$off(handlerId)
```

Arguments:

`handlerId` The unique id identifying the handler

Method `trigger()`**:** Trigger an event in the app. This will cause any handler attached to the event to be called. See the *The event cycle in fiery* vignette for more information.

Usage:

```
Fire$trigger(event, ...)
```

Arguments:

`event` The name of the event

`...` Arguments passed on to the handlers

Returns: A named list containing the return values of all handlers attached to the event

Method `send()`**:** Send a Websocket message to a client. Will trigger the `send` event.

Usage:

```
Fire$send(message, id)
```

Arguments:

`message` The message to send

`id` The id of the client to send to. If missing, the message will be send to all clients

Method `close_ws_con()`**:** Close a Websocket connection. Will trigger the `websocket-closed` event

Usage:

```
Fire$close_ws_con(id)
```

Arguments:

`id` The id of the client to close the websocket connection to

Method `serve_static()`**:** Serve a file or directory of files at a specified url path. Requests matching a file on the system never enters into the request loop but are served directly (and fast). Due to this, logging for these requests are also turned off

Usage:

```
Fire$serve_static(
  at,
  path,
  use_index = TRUE,
  fallthrough = FALSE,
  html_charset = "utf-8",
  headers = list(),
  validation = NULL
)
```

Arguments:

`at` The url path to listen to requests on

`path` The path to the file or directory on the file system

`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 for 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 `character(0)` (the default), then no validation check will be performed.

Method `exclude_static()`**:** Exclude a url path from serving static content. Only meaningful to exclude sub paths of path that are set to serve static content

Usage:

```
Fire$exclude_static(at)
```

Arguments:

`at` The url path to exclude from static serving. Request to this path will enter the normal request loop

Method `attach()`**:** Attach a plugin to the app. See the *Creating and using fiery plugins* vignette for more information

Usage:

```
Fire$attach(plugin, ..., name = NULL, force = FALSE)
```

Arguments:

`plugin` The plugin to attach

`...` Arguments to pass into the plugins `on_attach()` method

`name` Optional name for the plugin. If omitted `plugin$name` will be used instead

`force` If the plugin has already been attached an error is thrown, unless `force = TRUE` which tells the app to reattach it

Method `has_plugin()`**:** Check if the app has a plugin attached

Usage:

```
Fire$has_plugin(name)
```

Arguments:

`name` The name of the plugin

Returns: A boolean indicating if the given plugin is already attached

Method `header()`**:** Add a global http header that will be applied to all responses

Usage:

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

Arguments:

`name` The name of the header

`value` The value of the header. Use `NULL` to remove the global header

Method `set_data()`**:** Add data to the global data store

Usage:

```
Fire$set_data(name, value)
```

Arguments:

`name` The name identifying the data

`value` The data to add

Method `get_data()`**:** Retrieve data from the global data store

Usage:

```
Fire$get_data(name)
```

Arguments:

`name` The name identifying the data

Returns: The data requested. Returns `NULL` if the store does not contain the requested data

Method `remove_data()`**:** Remove data from the global data store

Usage:

```
Fire$remove_data(name)
```

Arguments:

`name` The name identifying the data to be removed

Method `time()`**:** Add a timed evaluation that will be evaluated after the given number of seconds.

Usage:

```
Fire$time(expr, then, after, loop = FALSE)
```

Arguments:

`expr` The expression to evaluate when the time has passed

`then` A handler to call once `expr` has been evaluated

`after` The time in second to wait before evaluating `expr`

`loop` Should `expr` be called repeatedly with the interval given by `after`

Returns: A unique id identifying the handler

Method `remove_time()`**:** Remove a timed evaluation

Usage:

```
Fire$remove_time(id)
```

Arguments:

`id` The unique id identifying the handler

Method `delay()`**:** Add a delayed evaluation to be evaluated immediately at the end of the loop cycle.

Usage:

```
Fire$delay(expr, then)
```

Arguments:

`expr` The expression to evaluate at the end of the cycle

`then` A handler to call once `expr` has been evaluated

Returns: A unique id identifying the handler

Method `remove_delay()`**:** Remove a delayed evaluation

Usage:

```
Fire$remove_delay(id)
```

Arguments:

`id` The unique id identifying the handler

Method `async()`**:** **[Deprecated]** Add an asynchronous evaluation to be evaluated in another process without blocking the server. This function has been deprecated in favor of using your own async framework of choice, e.g. mirai or promises

Usage:

```
Fire$async(expr, then)
```

Arguments:

`expr` The expression to evaluate at the end of the cycle

`then` A handler to call once `expr` has been evaluated

Returns: A unique id identifying the handler

Method `remove_async()`**:** Remove an async evaluation

Usage:

```
Fire$remove_async(id)
```

Arguments:

`id` The unique id identifying the handler

Method `set_client_id_converter()`**:** Sets the function that converts an HTTP request into a specific client id

Usage:

```
Fire$set_client_id_converter(converter)
```

Arguments:

`converter` A function with the argument `request`

Method `set_logger()`**:** Sets the logging function to use

Usage:

```
Fire$set_logger(logger)
```

Arguments:

`logger` A function with the arguments `event`, `message`, `request`, and `...`

Method `log()`**:** Log a message with the logger attached to the app. See loggers for build in functionality

Usage:

```
Fire$log(
  event,
  message,
  request = NULL,
  ...,
```

```
  .logcall = sys.call(),
  .topcall = sys.call(-1),
  .topenv = parent.frame()
)
```

Arguments:

`event` The event associated with the message

`message` The message to log

`request` The `Request` object associated with the message, if any.

`...` Additional arguments passed on to the logger.

`.logcall` The call that send the log request

`.topcall` The call in which `.logcall` is called from

`.topenv` The environment associated with `.topcall`

Method `is_running()`**:** Test if an app is running

Usage:

```
Fire$is_running()
```

Method `safe_call()`**:** Evaluate an expression safely, logging any errors, warnings, or messages that bubbles up

Usage:

```
Fire$safe_call(expr, request = NULL)
```

Arguments:

`expr` An expression to evaluate

`request` The request under evaluation, if any. Used in logging

Returns: The value of the expression. If an error is caught, the condition object is returned instead

Method `test_request()`**:** Send a request directly to the request logic of a non-running app. Only intended for testing the request logic

Usage:

```
Fire$test_request(request)
```

Arguments:

`request` The request to send

Method `test_header()`**:** Send a request directly to the header logic of a non-running app. Only intended for testing the request logic

Usage:

```
Fire$test_header(request)
```

Arguments:

`request` The request to send

Method `test_message()`**:** Send a message directly **to** the message logic of a non-running app. Only intended for testing the websocket logic

Usage:

```
Fire$test_message(request, binary, message, withClose = TRUE)
```

Arguments:

`request` The request to use to establish the connection

`binary` Is the message send in binary or character format

`message` The message to send. If `binary = FALSE` a character vector, if `binary = TRUE` a raw vector

`withClose` Should the websocket connection be closed at the end by the client

Method `test_websocket()`**:** Send a message directly **from** a non-running app. Only intended for testing the websocket logic

Usage:

```
Fire$test_websocket(request, message, close = TRUE)
```

Arguments:

`request` The request to use to establish the connection

`message` The message to send from the app

`close` Should the websocket connection be closed at the end by the server

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

Usage:

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

Arguments:

`deep` Whether to make a deep clone.

Examples

```
# Create a New App
app <- Fire$new(port = 4689)

# Setup the data every time it starts
app$on('start', function(server, ...) {
    server$set_data('visits', 0)
    server$set_data('cycles', 0)
})

# Count the number of cycles
app$on('cycle-start', function(server, ...) {
    server$set_data('cycles', server$get_data('cycles') + 1)
})

# Count the number of requests
app$on('before-request', function(server, ...) {
    server$set_data('visits', server$get_data('visits') + 1)
})

# Handle requests
app$on('request', function(server, ...) {
```

```
    list(
        status = 200L,
        headers = list('Content-Type' = 'text/html'),
        body = paste('This is indeed a test. You are number', server$get_data('visits'))
    )
})

# Show number of requests in the console
app$on('after-request', function(server, ...) {
    message(server$get_data('visits'))
    flush.console()
})

# Terminate the server after 300 cycles
app$on('cycle-end', function(server, ...) {
    if (server$get_data('cycles') > 300) {
        message('Ending...')
        flush.console()
        server$extinguish()
    }
})

# Be polite
app$on('end', function(server) {
    message('Goodbye')
    flush.console()
})

## Not run:
app$ignite(showcase = TRUE)

## End(Not run)
```

`loggers` *App Logging*

Description

`fiery` has a build in logging mechanism that lets you capture event information however you like. Every user-injested warnings and errors are automatically captured by the logger along with most system errors as well. `fiery` tries very hard not to break due to faulty app logic. This means that any event handler error will be converted to an error log without `fiery` stopping. In the case of request handlers a 500L response will be send back if any error is encountered.

Usage

```
logger_void(...)

logger_null()
```

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

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

logger_otel(format = "{message}")

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

logger_logger(default_level = "INFO")

common_log_format

combined_log_format
```

Arguments

`...`	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'))`.
`format`	A glue-like string specifying the format of the log entry. Only the variables `time`, `event`, and `message` are available and must be given verbatim
`file`	A file or connection to write to
`default`	A catch-all logger for use with events not defined in `...`
`default_level`	The log level to use for events that are not `request`, `websocket`, `message`, `warning`, or `error`

Format

An object of class `character` of length 1.

An object of class `character` of length 1.

Setting a logger

By default, `fiery` uses `logger_null()` which forwards warning and error messages to `stderr()` and ignores any other logging events. To change this behavior, set a different logger using the `set_logger()` method:

```
app$set_logger(logger)
```

where `logger` is a function taking at least the following arguments: `event`, `message`, `request`, `time`, and `....`

`fiery` comes with some additional loggers, which either writes all logs to a file or to the console. A new instance of the file logger can be created with `logger_file(file)`:

```
app$set_logger(logger_file('fiery_log.log'))
```

A new instance of the console logger can be create with `logger_console()`:

```
app$set_logger(logger_console())
```

Both functions takes a `format` a argument that lets you customise how the log is written. Furthermore the console logger will style the logs with colour coding depending on the content if the console supports it.

As a last possibility it is possible to use different loggers dependent on the event by using the switch logger:

```
app$set_logger(logger_switch(warning =,
                             error = logger_file('errors.log),
                             default = logger_file('info.log')))
```

Automatic logs

`fiery` logs a number of different information by itself describing its operations during run. The following events are send to the log:

start Will be send when the server starts up

resume Will be send when the server is resumed

stop Will be send when the server stops

request Will be send when a request has been handled. The message will contain information about how long time it took to handle the request or if it was denied.

websocket Will be send every time a WebSocket connection is established or closed as well as when a message is received or send

message Will be send every time a message is emitted by an event handler or delayed execution handler

warning Will be send everytime a warning is emitted by an event handler or delayed execution handler

error Will be send everytime an error is signaled by an event handler or delayed execution handler. In addition some internal functions will also emit error event when exceptions are encountered

By default only *message*, *warning* and *error* events will be logged by sending them to the error stream as a `message()`.

Access Logs

Of particular interest are logs that detail requests made to the server. These are the `request` events detailed above. There are different standards for how requests are logged. `fiery` uses the *Common Log Format* by default, but this can be modified by setting the `access_log_format` field to a glue expression that has access to the following variables:

`start_time` The time the request was received

`end_time` The time the response was send back

`request` The `Request` object

`response` The `Response` object

`id` The client id

To change the format:

```
app$access_log_format <- combined_log_format
```

Custom logs

Apart from the standard logs described above it is also possible to send messages to the log as you please, e.g. inside event handlers. This is done through the `log()` method where you at the very least specify an event and a message. You can also log information using the standard exception mechanism. Errors, warnings and messages are all caught and logged and since exception carry more information than a log string, the logging can be richer.

An example of using `log()` in a handler could be:

```
app$on('header', function(server, id, request) {
  server$log('info', paste0('request from ', id, ' received'), request)
})
```

Which would log the timepoint the headers of a request has been recieved.

`session_id_cookie` *Use a session cookie to store the id of the session*

Description

This function constructs an ID extractor for incoming requests, the return value of which will be passed to the id argument of request, header, message etc handlers. By default, fiery uses this with default arguments.

Usage

```
session_id_cookie(name = "fiery_id", secure = FALSE)
```

Arguments

`name`	The name of the cookie to store the session id in
`secure`	Should the session id only be send over HTTPS? Setting this to TRUE will require setting up a proxy manager with SSL support in front of your fiery server.

Details

A session id is looked for in the cookie matching the `name` arg. If it is found then the value of the cookie is returned. If it is not found then an id is generated with `reqres::random_key()` and attached to the response as a cookie with the given name. The cookie is set to HTTP only and Strict same site policy. Depending on `secure` it also sets it to only be transferred over HTTPS.

Value

A unary function taking a Request object and returning an ID for it

Note

Session ID should not be considered as authentication. If you are handling sensitive data you should consider a more secure way of identifying users across requests.

Index