# Package 'fiery'

December 15, 2020

**Type** Package

**Title** A Lightweight and Flexible Web Framework

**Version** 1.1.3

**Maintainer** Thomas Lin Pedersen <thomasp85@gmail.com>

**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

**Encoding** UTF-8

**Imports** R6, assertthat, httpuv, uuid, utils, stringi, future, later, stats, reqres, glue, crayon

**Collate** 'loggers.R' 'aaa.R' 'HandlerStack.R' 'Fire.R' 'FutureStack.R' 'fake_request.R' 'fiery-package.R'

**RoxygenNote** 7.1.1

**Suggests** testthat, covr, knitr, rmarkdown

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

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

**VignetteBuilder** knitr

**NeedsCompilation** no

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

**Repository** CRAN

**Date/Publication** 2020-12-15 22:50:06 UTC

# R topics documented:



---

`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:

### Usage

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

### Arguments

| | |
|---|---|
| `host` | A string overriding the default host (see the *Fields* section below) |
| `port` | An integer overriding the default port (see the *Fields* section below) |

*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.

## Fields

`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` 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-bocking server (defaults to 1)

`trigger_dir` A valid folder where trigger files can be put when running a blocking server (defaults to `NULL`)

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

`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*

## Methods

`ignite(block = TRUE, showcase = FALSE, ...)` Begins the server, either blocking the console if `block = TRUE` or not. If `showcase = TRUE` a browser window is opened directing at the server address. `...` will be redirected to the `start` handler(s)

`start(block = TRUE, showcase = FALSE, ...)` A less dramatic synonym of for `ignite()`

`reignite(block = TRUE, showcase = FALSE, ...)` As `ignite` but additionally triggers the `resume` event after the `start` event

`resume(block = TRUE, showcase = FALSE, ...)` Another less dramatic synonym, this time for `reignite()`

`extinguish()` Stops a running server

`stop()` Boring synonym for `extinguish()`

`is_running()` Check if the server is currently running

`on(event, handler, pos = NULL)` Add a `handler` function to to an `event` at the given position (`pos`) in the handler stack. Returns a string uniquely identifying the handler. See the event vignette for more information.

`off(handlerId)` Remove the handler tied to the given `id`

`trigger(event, ...)` Triggers an `event` passing the additional arguments to the potential handlers

`send(message, id)` Sends a websocket `message` to the client with the given `id`, or to all connected clients if `id` is missing

`log(event, message, request, ...)` Send a `message` to the logger. The `event` defines the type of message you are passing on, while `request` is the related `Request` object if applicable.

`close_ws_con(id)` Closes the websocket connection started from the client with the given `id`, firing the `websocket-closed` event

`attach(plugin, ..., force = FALSE)` Attaches a `plugin` to the server. See the plugin vignette for more information. Plugins can only get attached once unless `force = TRUE`

`has_plugin(name)` Check whether a plugin with the given `name` has been attached

`header(name, value)` Add a global `header` to the server that will be set on all responses. Remove by setting `value = NULL`

`set_data(name, value)` Adds data to the servers internal data store

`get_data(name)` Extracts data from the internal data store

`remove_data(name)` Removes the data with the given `name` from the internal data store

`time(expr, then, after, loop = FALSE)` Add a timed evaluation (`expr`) that will be evaluated after the given number of seconds (`after`), potentially repeating if `loop = TRUE`. After the expression has evaluated the `then` function will get called with the result of the expression and the server object as arguments.

`remove_time(id)` Removes the timed evaluation identified by the `id` (returned when adding the evaluation)

`delay(expr, then)` Similar to `time()`, except the `expr` is evaluated immediately at the end of the loop cycle (see here for detailed explanation of delayed evaluation in fiery).

`remove_delay(id)` Removes the delayed evaluation identified by the `id`

`async(expr, then)` As `delay()` and `time()` except the expression is evaluated asynchronously. The progress of evaluation is checked at the end of each loop cycle

`remove_async(id)` Removes the async evaluation identified by the `id`. The evaluation is not necessarily stopped but the then function will not get called.

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

`set_logger(logger)` Sets the function that takes care of logging

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

`clone()` Create a copy of the full `Fire` object and return that

## 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$close_ws_con()`
- `Fire$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$is_running()
- Fire$test_request()
- Fire$test_header()
- Fire$test_message()
- Fire$test_websocket()
- Fire$clone()

**Method** `new()`**:**

*Usage:*

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

**Method** `format()`**:**

*Usage:*

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

**Method** `ignite()`**:**

*Usage:*

```
Fire$ignite(block = TRUE, showcase = FALSE, ..., silent = FALSE)
```

**Method** `start()`**:**

*Usage:*

```
Fire$start(block = TRUE, showcase = FALSE, ..., silent = FALSE)
```

**Method** `reignite()`**:**

*Usage:*

```
Fire$reignite(block = TRUE, showcase = FALSE, ..., silent = FALSE)
```

**Method** `resume()`**:**

*Usage:*

```
Fire$resume(block = TRUE, showcase = FALSE, ..., silent = FALSE)
```

**Method** `extinguish()`**:**

*Usage:*

```
Fire$extinguish()
```

**Method** `stop()`**:**

*Usage:*

```
Fire$stop()
```

**Method** `on()`**:**

*Usage:*

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

**Method** `off()`**:**

*Usage:*

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

**Method** `trigger()`**:**

*Usage:*

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

**Method** `send()`**:**

*Usage:*

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

**Method** `close_ws_con()`**:**

*Usage:*

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

**Method** `attach()`**:**

*Usage:*

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

**Method** `has_plugin()`**:**

*Usage:*

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

**Method** `header()`**:**

*Usage:*

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

**Method** `set_data()`**:**

*Usage:*

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

**Method** `get_data()`**:**

*Usage:*

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

**Method** `remove_data()`**:**

*Usage:*

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

**Method** `time()`**:**

*Usage:*

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

**Method** `remove_time()`**:**

*Usage:*

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

**Method** `delay()`**:**

*Usage:*

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

**Method** `remove_delay()`**:**

*Usage:*

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

**Method** `async()`**:**

*Usage:*

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

**Method** `remove_async()`**:**

*Usage:*

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

**Method** `set_client_id_converter()`**:**

*Usage:*

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

**Method** `set_logger()`**:**

*Usage:*

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

**Method** `log()`**:**

*Usage:*

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

**Method** `is_running()`**:**

*Usage:*

```
Fire$is_running()
```

**Method** `test_request()`**:**

*Usage:*

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

**Method** `test_header()`**:**

*Usage:*

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

**Method** `test_message()`**:**

*Usage:*

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

**Method** `test_websocket()`**:**

*Usage:*

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

**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_null()

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

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

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

common_log_format

combined_log_format
```

## Arguments

| | |
|---|---|
| format | A glue string specifying the format of the log entry |
| file | A file or connection to write to |
| ... | 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 ... |

## 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 recieved

`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. In general it is better to send messages through `log()` rather than with `warning()` and `stop()` even though the latters will eventually be caught, as it gives you more control over the logging and what should happen in the case of an exception.

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.

# Index