Package 'fiery'

October 22, 2018

Type Package

Title A Lightweight and Flexible Web Framework

Version 1.1.1

Date 2018-10-22

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' 'delay_doc.R' 'event_doc.R' 'fake_request.R' 'fiery-package.R' 'plugin_doc.R'

RoxygenNote 6.1.0

Suggests testthat, covr

URL https://github.com/thomasp85/fiery

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

NeedsCompilation no

Author Thomas Lin Pedersen [aut, cre]

Repository CRAN

Date/Publication 2018-10-22 10:00:04 UTC

R topics documented:

`fiery-package` *fiery: A Lightweight and Flexible Web Framework*

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.

Details

`fiery` is a lightweight and flexible framework for web servers build on top of the httpuv package. The framework is largely event-based, letting the developer attach handlers to life-cycle events as well as defining and triggering their own events. This approach to development is common in JavaScript, but might feel foreign to R developers. Thankfully it is a rather simple concept that should be easy to gradually begin to use to greater and greater effect.

Read more:

- Creation of the server object, along with all its methods and fields, is described in the documentation of the `Fire` class.
- An overview of the event model, along with descriptions of the predefined life-cycle events and custom events can be found in the events documentation.
- A description of the `fiery` plugin interface and how to develop your own plugins is laid out in the plugins documentation

Author(s)

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

See Also

Useful links:

- https://github.com/thomasp85/fiery
- Report bugs at https://github.com/thomasp85/fiery/issues

delay_doc *Delaying code execution in Fiery*

Description

R, and thus `fiery`, is single threaded, meaning that every request must be handled one at a time. Because of this it is of utmost importance to keep the computation time for each request handling as low as possible so that the server does not become unresponsive. Still, sometimes you may need to perform long running computations as part of the server functionality. `fiery` comes with three different facilities for this, each with its own use case. All of them are build on top of future.

General format

All three methods have the same general API. They can recieve an expression to evaluate, as well as a `then` function to call once the evaluation eventually completes. The `then` function will recieve the result of the provided expression as well as the server itself. In general, any code that works on the server should be handled by the `then` function as the expression will not necessarily have access to the current environment. Thus, the expression should be as minimal as possible while still containing the heavy part of the calculations, while the `then` function should be used to act upon the result of the expression.

The general format is thus (using `delay()` as an example):

```
app$delay({
  # Heavy calculation
}, then = function(res, server) {
  # Do something with 'res' (the result of the expression) and 'server' the
  # server object itself
})
```

Pushing execution to the end of a cycle

If it is important to achieve a fast response time, but server congestion is of lesser concern (the server might be used for a local app with only one user at a time), the `delay()` method can be used to push the evaluation of long running computation to the end of the current cycle. It will of course not be possible to return the result of the computation as part of the response, but e.g. a `202` response can be returned instead indicating that the request is being processed. In that way the client can act accordingly without appearing frozen. An alternative if a lengthy POST request is recieved is to return `303` with a reference to the URL where the result can be recieved.

Executing in another process

If long running computations are needed and congestion is an issue it does not help to simply push back execution to the end of the cycle as this will block requests while the code is evaluating. Instead it is possible to use the `async()` method to evaluate the expression in another thread. This method uses `future::multiprocess()` to evaluate the expression and may thus fork the current R process if supported (Unix systems) or start another R session (Windows). At the end of each cycle all async evaluations are checked for completion, and if completed the `then` function will be called with the result. If the async evaluation is not completed it will continue to churn.

Executing after a time interval

If code is meant to be evaluated after a certain amount of time has passed, use the `time()` method. In addition to `expr` and `then`, `time()` takes two additional arguments: `after` (the time in seconds to wait before evaluation) and `loop` (whether to repeat the timed evaluation after completion). Using `loop = TRUE` it is e.g. possible to continually check for state changes on the server and e.g. run some specific code if new files appear in a directory. In the end of each cycle all timed expressions will be checked for whether they should be evaluated and run if their specific time interval has passed.

Error handling

As both the expression and `then` function might throw errors they are evaluated in a safe context and any errors that might occur will be send to the server log without affecting other waiting evaluations.

`event_doc` *Event Overview*

Description

fiery is using an event-based model to allow you to program the logic. During the life cycle of an app a range of different events will be triggered and it is possible to add event handlers to these using the `on()` method. An event handler is simply a function that will get called every time an event is fired. Apart from the predefined life cycle events it is also possible to trigger custom events using the `trigger()` method. Manual triggering of life cycle events is not allowed.

Life cycle Events

Following is a list of all life cycle events. These cannot be triggered manually, but is fired as part of the normal lifetime of a `fiery` server:

start Will trigger once when the app is started but before it is running. The handlers will receive the app itself as the `server` argument as well as any argument passed on from the `ignite()` method. Any return value is discarded.

resume Will trigger once after the start event if the app has been started using the `reignite()` method. The handlers will receive the app itself as the `server` argument as well as any argument passed on from the `reignite()` method. Any return value is discarded.

end Will trigger once after the app is stopped. The handlers will receive the app itself as the `server` argument. Any return value is discarded.

cycle-start Will trigger in the beginning of each loop, before the request queue is flushed. The handlers will receive the app itself as the `server` argument. Any return value is discarded.

cycle-end Will trigger in the end of each loop, after the request queue is flushed and all delayed, timed, and asynchronous calls have been executed. The handlers will receive the app itself as the `server` argument. Any return value is discarded.

header Will trigger every time the header of a request is received. The return value of the last called handler is used to determine if further processing of the request will be done. If the return value is `TRUE` the request will continue on to normal processing. If the return value is `FALSE` the response will be send back and the connection will be closed without retrieving the payload. The handlers will receive the app itself as the `server` argument, the client id as the `id` argument and the request object as the `request` argument

before-request Will trigger prior to handling of a request (that is, every time a request is received unless it is short-circuited by the `header` handlers). The return values of the handlers will be passed on to the request handlers and can thus be used to inject data into the request handlers (e.g. session specific data). The handlers will receive the app itself as the `server` argument, the client id as the `id` argument and the request object as the `request` argument

request Will trigger after the `before-request` event. This is where the main request handling is done. The return value of the last handler is send back to the client as response. If no handler is registered a `404` error is returned automatically. If the return value is not a valid response, a `500` server error is returned instead. The handlers will receive the app itself as the `server` argument, the client id as the `id` argument, the request object as the `request` argument, and the list of values created by the before-event handlers as the `arg_list` argument.

after-request Will trigger after the `request` event. This can be used to inspect the response (but not modify it) before it is send to the client. The handlers will receive the app itself as the `server` argument, the client id as the `id` argument, the request object as the `request` argument, and the response as the `response` argument. Any return value is discarded.

before-message This event is triggered when a websocket message is received. As with the `before-request` event the return values of the handlers are passed on to the `message` handlers. Specifically if a `'binary'` and `'message'` value is returned they will override the original values in the `message` and `after-message` handler arguments. This can e.g. be used to decode the message once before passing it through the `message` handlers. The `before-message` handlers will receive the app itself as the `server` argument, the client id as the `id` argument, a flag indicating whether the message is binary as the `binary` argument, the message itself as the `message` argument, and the request object used to establish the connection with the client as the `request` argument.

message This event is triggered after the `before-message` event and is used for the primary websocket message handling. As with the `request` event, the handlers for the `message` event receives the return values from the `before-message` handlers which can be used to e.g. inject session specific data. The message handlers will receive the app itself as the `server` argument, the client id as the `id` argument, a flag indicating whether the message is binary as the `binary` argument, the message itself as the `message` argument, the request object used to establish the connection with the client as the `request` argument, and the values returned by the before-message handlers as the `arg_list` argument. Contrary to the `request` event the return values of the handlers are ignored as websocket communication is bidirectional

after-message This event is triggered after the `message` event. It is provided more as an equivalent to the `after-request` event than out of necessity as there is no final response to inspect and handler can thus just as well be attached to the `message` event. For clear division of server logic, message specific handlers should be attached to the `message` event, whereas general handlers should, if possible, be attached to the `after-message` event. The `after-message` handlers will receive the app itself as the `server` argument, the client id as the `id` argument, a flag indicating whether the message is binary as the `binary` argument, the message itself as

the `message` argument, and the request object used to establish the connection with the client as the `request` argument.

send This event is triggered after a websocket message is send to a client. The handlers will receive the app itself as the `server` argument, the client id as the `id` argument and the send message as the `message` argument. Any return value is discarded.

websocket-closed This event will be triggered every time a websocket connection is closed. The handlers will receive the app itself as the `server` argument, the client id as the `id` argument and request used to establish the closed connection as the `request` argument. Any return value is discarded.

Custom Events

Apart from the predefined events, it is also possible to trigger and listen to custom events. The syntax is as follows:

```
# Add a handler to the 'new-event' event
id <- app$on('new-event', function() {
  message('Event fired')
})

# Trigger the event
app$trigger('new-event')

# Remove the handler
app$off(id)
```

Additional parameters passed on to the `trigger()` method will be passed on to the handler. There is no limit to the number of handlers that can be attached to custom events. When an event is triggered they will simply be called in the order they have been added. Triggering a non-existing event is not an error, so plugins are free to fire off events without worrying about whether handlers have been added.

Triggering Events Externally

If a `fiery` server is running in blocking mode it is not possible to communicate with it using the `trigger()` method. Instead it is possible to assign a directory to look in for event trigger instructions. The trigger directory is set using the `trigger_dir` field, e.g.:

```
app$trigger_dir <- '/some/path/to/dir/'
```

Events are triggered by placing an `rds` file named after the event in the trigger directory. The file must contain a list, and the elements of the list will be passed on as arguments to the event handlers. After the event has been triggered the file will be deleted. The following command will trigger the `external-event` on a server looking in `'/some/path/to/dir/'`:

```
saveRDS(list(arg1 = 'test'), '/some/path/to/dir/external-event.rds')
```

See Also

Fire describes how to create a new server

plugins describes how to use plugins to modify the server

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 documentation. fiery servers can be modified directly or by attaching plugins. As with events, plugins has its own documentation.

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 documentation 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 documentation 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

See Also

events describes how the server event cycle works

plugins describes how to use plugins to modify the server

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
default	A catch-all logger for use with events not defined in ...

Format

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.

`plugin_doc` *Plugin Interface*

Description

In order to facilitate encapsulate functionality that can be shared between `fiery` servers `fiery` implements a plugin interface. Indeed, the reason why `fiery` is so minimal in functionality is because it is intended as a foundation for separate plugins that can add convenience and power. This approach allows `fiery` itself to remain unopinionated and flexible.

Using Plugins

Plugins are added to a `Fire` object using the `attach()` method. Any parameters passed along with the plugin to the `attach()` method will be passed on to the plugins `on_attach()` method (see below).

```
app$attach(plugin)
```

Creating Plugins

The `fiery` plugin specification is rather simple. A plugin is either a list or environment (e.g. a RefClass or R6 object) with the following elements:

`on_attach(server, ...)` A function that will get called when the plugin is attached to the server. It is passed the server object as the first argument along with any arguments passed to the `attach()` method.

`name` A string giving the name of the plugin

`require` **Optional** A character vector giving names of other plugins that must be attached for this plugin to work

Apart from this, the list/environment can contain anything you desires. For an example of a relatively complex plugin, have a look at the source code for the routr package.

Accessing Plugins

When a plugin is attached to a `Fire` object, two things happens. First, the `on_attach()` function in the plugin is called modifying the server in different ways, then the plugin object is saved internally, so that it can later be retrieved. All plugins are accessible in the `plugins` field under the name of the plugin. This is useful for plugins that modifies other plugins, or are dependent on functionality in other plugins. A minimal example of a plugin using another plugin could be:

```
plugin <- list(
  on_attach = function(server) {
    router <- server$plugins$request_routr
    route <- Route$new()
    route$add_handler('all', '*', function(request, response, arg_list, ...) {
      message('Hello')
      TRUE
    })
    router$add_route(route, 1)
  },
  name = 'Hello_plugin',
  require = 'request_routr'
)
```

The `Hello_plugin` depends on the routr plugin for its functionality as it modifies the request router to always say hello when processing requests. If the `reques_routr` plugin has not already been attached it is not possible to use the `Hello_plugin` plugin.

It is also possible to have a soft dependency to another plugin, by not listing it in `require` and instead use the `has_plugin()` method in the server to modify the behaviour of the plugin. We could rewrite the `Hello_plugin` to add the routr plugin by itself if missing:

```
plugin <- list(
  on_attach = function(server) {
    if (!server$has_plugin('request_routr')) {
      server$attach(RouteStack$new())
    }
    router <- server$plugins$request_routr
    route <- Route$new()
    route$add_handler('all', '*', function(request, response, arg_list, ...) {
      message('Hello')
      TRUE
    })
    router$add_route(route, 1)
  },
  name = 'Hello_plugin2'
)
```

See Also

`Fire` describes how to create a new server

events describes how the server event cycle works

`random_port` *Select a random safe port*

Description

This is a small utility function to get random safe ports to run your application on. It chooses a port within the range that cannot be registeret to IANA and thus is safe to assume are not in use.

Usage

```
random_port()
```

Value

An integer in the range 49152-65535

Examples

```
random_port()
```

Index