gaffer Documentation

Release 0.5.2

Author

October 16, 2015

Contents

1 Gaffer **1**
- 1.1 Features . . . 1
- 1.2 Contents: . . . 2

2 Indices and tables **29**

Python Module Index **31**

CHAPTER 1

Gaffer

Control, Watch and Launch your applications and jobs over HTTP.

Gaffer is a set of Python modules and tools to easily maintain and interact with your applications or jobs launched on different machines over HTTP and websockets.

It promotes distributed and decentralized topologies without single points of failure, enabling fault tolerance and high availability.

1.1 Features

- RESTful HTTP Api
- Websockets and SOCKJS support to interact with a gaffer node from any browser or SOCKJS client.
- Framework to manage and interact your applications and jobs on differerent machines
- Server and Command Line tools to manage and interract with your processes
- manages topology information. Clients query gaffer_lookupd to discover gaffer nodes for a specific job or application.
- Possibility to interact with STDIO and PIPES to interact with your applications and processes
- Subscribe to process statistics per process or process templates and get them in quasi RT.
- Procfile applications support (see Gaffer) but also JSON config support.
- Supervisor-like features.
- Fully evented. Use the libuv event loop using the pyuv library
- Flapping: handle cases where your processes crash too much
- Easily extensible: add your own endpoint, create your client, embed gaffer in your application, ...
- Compatible with python 2.7x, 3.x

Note: gaffer source code is hosted on Github

1.2 Contents:

1.2.1 Getting started

This tutorial exposes the usage of gaffer as a tool. For a general overview or how to integrate it in your application you should read the overview page.

Introduction

Gaffer allows you to launch OS processes and supervise them. 3 command line tools allows you to use it for now:

- Gafferd is the process supervisor and should be launched first before to use other tools.
- Gaffer is a Procfile application manager and allows you to load your Procfile applications in gafferd and watch their status.
- `gafferctl` is a more generic tooll than gaffer and is more admin oriented. It allows you to setup any process templates and manage your processes. You can also use it to watch the activity in gafferd (process activity or general activity)

A process template is the way you describe the launch of an OS process, how many you want to launch on startup, how many time you want to restart it in case of failures (flapping).... A process template can be loaded using any tool or on *gafferd* startup using its configuration file.

Workflow

To use gaffer tools you need to:

1. First launch gafferd
2. use either gaffer or gafferctl to manage your processes

Launch gafferd

For more informations of gafferd go on its documentation page .

To launch gafferd run the following command line:

```
$ gafferd -c /path/to/gaffer.ini
```

If you want to launch custom plugins with gafferd you can also set the path to them:

```
$ gafferd -c /path/to/gaffer.ini -p /path/to/plugin
```

Note: default plugin path is relative to the user launching gaffer and is set to *~/.gaffer/plugins*.

Note: To launch it in daemon mode use the `--daemon` option.

Then with the default configuration, you can check if gafferd is alive

The configuration file

The configuration file can be used to set the global configuration of gafferd, setup some processes and webhooks.

Note: Since the configuration is passed to the plugin you can also use this configuration file to setup your plugins.

Here is a simple example of a config to launch the dumy process from the example folder:

```
[process:dummy]
cmd = ./dummy.py
numprocesses = 1
redirect_output = stdout, stderr
```

Note: Process can be grouped. You can then start and stop all processes of a group and see if a process is member of a group using the HTTP api. (sadly this is not yet possible to do it using the command line).

For example if you want dummy be part of the group test, then `[process:dummy]` will become `[process:test:dummy]` . A process template as you can see can only be part of one group.

Groups are useful when you want to manage a configuration for one application or processes / users.

Each process section should be prefixed by *process:*. Possible parameters are:

- **cmd**: the full command line to launch. eg. `./dummy.p¨`
- **args**: arguments to pass as a string. eg. `–some value --option=a`
- **cwd**: path to working directorty
- **uid**: user name or id used to execute the process
- **gid**: group name or id used to execute the process
- **detach**: if you wnt to completely detach the process from gafferd (gaffer will still continue to supervise it)
- **shell**: The process is executed in a shell (unix only)
- **flapping**: flapping rule. eg. *2, 1., 7., 5* which means attempts=2, window=1., retry_in=7., max_retry=5
- **redirect_input**: to allows you to interract with stdin
- **redirect_output**: to watch both stdout & stderr. output names can be whatever you cant. For example you. eg. `redirect_output = mystdout, mystderr` stdout will be labelled *mysdtout* in this case.
- **graceful_timeout**: time to wait before definitely kill a process. By default 30s. When killing a process, gaffer is first sending a `SIGTERM` signal then after a graceful timeout if the process hasn't stopped by itself send a `SIGKILL` signal. It allows you to handle the way your process will stop.
- **os_env**: true or false, to pass all operating system variables to the process environment.
- **priority**: Integer. Allows you to fix the order in which gafferd will start the processes. 0 is the highest priority. By default all processes have the same order.

Sometimes you also want to pass a custom environnement to your process. This is done by creating a special configuration section named `env:processname`. Each environmenets sections are prefixed by `env:`. For example to pass a special PORT environment variable to dummy:

```
[env:dummy]
port = 80
```

All environment variables key are passed in uppercase to the process environment.

Manage your Procfile applications

The **gaffer** command line tool is an interface to the gaffer HTTP api and include support for loading/unloading Procfile applications, scaling them up and down, ...

It can also be used as a manager for Procfile-based applications similar to foreman but using the gaffer framework. It is running your application directly using a Procfile or export it to a gafferd configuration file or simply to a JSON file that you could send to gafferd using the HTTP api.

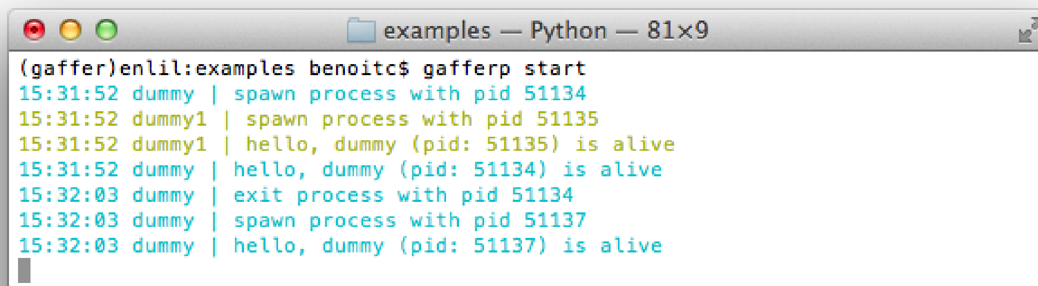
Example of use

For example using the following **Procfile**:

```
dummy: python -u dummy_basic.py
dummy1: python -u dummy_basic.py
```

You can launch all the programs in this procfile using the following command line:

```
$ gaffer start
```



Or load them on a gaffer node:

```
$ gaffer load
```

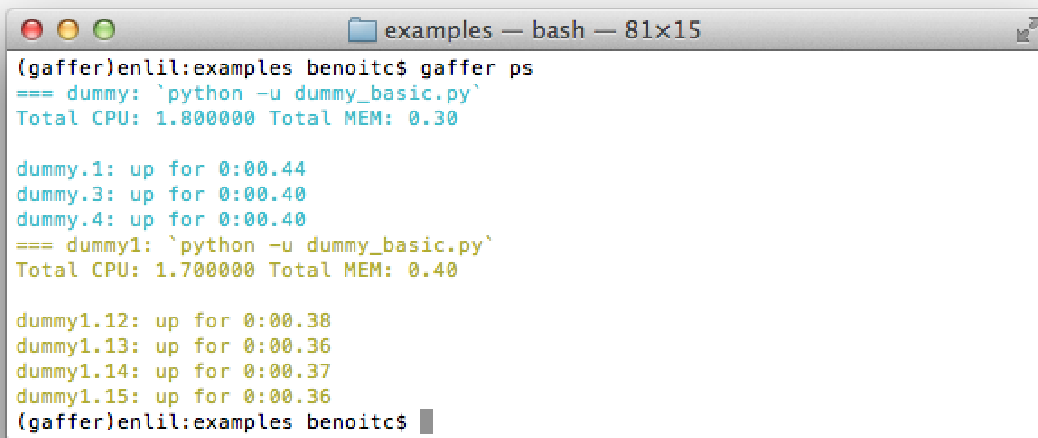
All processes in the Procfile will be then loaded to gafferd and started.

If you want to start a process with a specific environment file you can create a .env in he application folder (or use the command line option to tell to gaffer which one to use). Each environmennt variables are passed by lines. Ex:

```
PORT=80
```

and then scale them up and down:

```
$ gaffer scale dummy=3 dummy1+2
Scaling dummy processes... done, now running 3
Scaling dummy1 processes... done, now running 3
```



have a look on the Gaffer page for more informations about the commands.

Control gafferd with gafferctl

gafferctl can be used to run any command listed below. For example, you can get a list of all processes templates:

```
$ gafferctl processes
```

You can simply add a process using the `load` command:

```
$ gafferctl load_process ../test.json
$ cat ../test.json | gafferctl load_process -
$ gafferctl load_process - < ../test.json
```

test.json can be:

```
{
    "name": "somename",
    "cmd": "cmd to execute":
    "args": [],
    "env": {}
    "uid": int or "",
    "gid": int or "",
    "cwd": "working dir",
    "detach: False,
    "shell": False,
    "os_env": False,
    "numprocesses": 1
}
```

You can also add a process using the `add` command:

```
gafferctl add name inc
```

where `name` is the name of the process to create and `inc` the number of new OS processes to start.

To start a process run the following command:

```
$ gafferctl start name
```

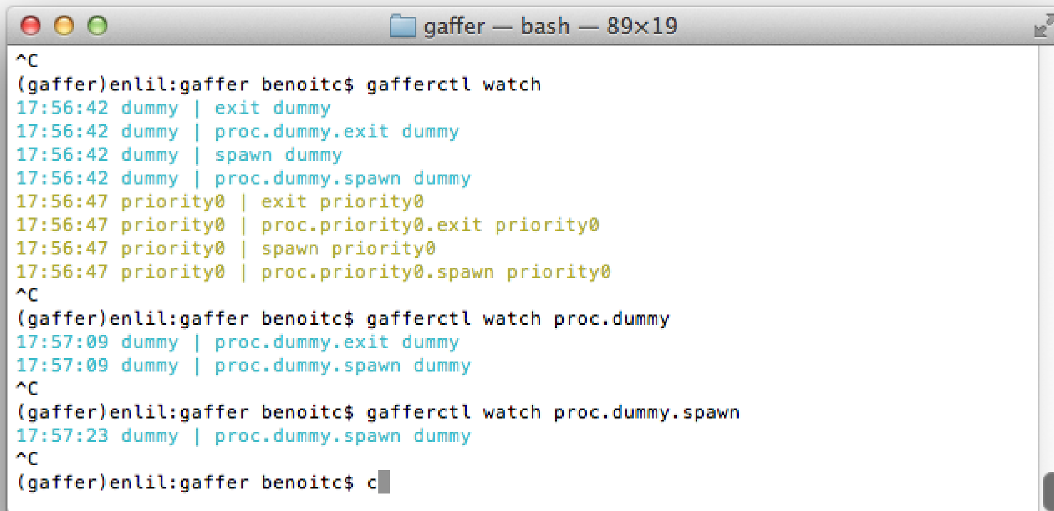
And stop it using the stop command.

To scale up a process use the `add` command. For example to increase the number of processes from 3:

```
$ gafferctl add name 3
```

To decrease the number of processes use the command `stop`/

The command `watch` allows you to watch changes n a local or remote gaffer node.



For more informations go on the `gafferctl` page.

Demo

1.2.2 Overview

Gaffer is a set of Python modules and tools to easily maintain and interact with your processes.

Depending on your needs you can simply use the gaffer tools (eventually extend them) or embed the gaffer possibilities in your own apps.

Design

Gaffer is internally based on an event loop using the libuv from Joyent via the pyuv binding

All gaffer events are added to the loop and processes asynchronously which make it pretty performant for handling & controlling multiple processes.

At the lowest level you will find the manager. A manager is responsible for maintaining live processes and managing actions on them:

- start/stop processes

- increase/decrease the number of instances of each process, via process templates
- add/remove process templates to manage

A process template describes the way a process will be launched and how many OS processes you want to handle for this template. This number can be changed dynamically.

Current properties for each template:

- **name**: name of the process (eg, 'django-server')
- **cmd**: program command, (eg. '/var/www/CMS/manage.py')
- **args**: the arguments for the command to run. Can be a list or a string. If **args** is a string, it's split using `shlex.split()`. Defaults to None.
- **env**: a mapping containing the environment variables the command will run with. Optional
- **uid**: int or str, user id
- **gid**: int or st, user group id,
- **cwd**: working dir
- **detach**: the process is launched but won't be monitored and won't exit when the manager is stopped.
- **shell**: boolean, run the script in a shell. (UNIX only),
- **os_env**: boolean, pass the os environment to the program
- **numprocesses**: int the number of OS processes to launch for this description
- **flapping**: a FlappingInfo instance or, if flapping detection should be used. flapping parameters are:
  - **attempts**: maximum number of attempts before we stop the process and set it to retry later
  - **window**: period in which we are testing the number of retry
  - **retry_in**: seconds, the time after we restart the process and try to spawn them
  - **max_retry**: maximum number of retry before we give up and stop the process.
- **redirect_output**: list of io to redict (max 2) this is a list of custom labels to use for the redirection. Ex: ["a", "b"] will redirect stdoutt & stderr and stdout events will be labeled "a"
- **redirect_input**: Boolean (False is the default). Set it if you want to be able to write to stdin.

The manager is also responsible of starting and stopping gaffer applications that you add to the manager to react to different events. A applicaton can fetch information from the manager and interact with it.

Running an application is done like this:

> # initialize the controller with the default loop loop = pyuv.Loop.default_loop() manager = Manager(loop=loop)
>
> # start the controller manager.start(applications=[HttpHandler()])
>
> .... # do smth
>
> manager.stop() # stop the controlller manager.run() # run the event loop

The HttpHandler application allows you to interact with gaffer via HTTP. It is used by the gafferd server which is able for now to load process templates via an ini files and maintain an HTTP endpoint which can be configured to be accessible on multiples interfaces and transports (tcp & unix sockets) .

Note: Only applications instances are used by the manager. It allows you to initialize them with your own settings.

Building your own application is easy, basically an application has the following structure:

```
class MyApplication(object):

    def __init__(self):
        # initialisation

    def start(self, loop, manager):
        # this method is called by the manager to start the controller

    def stop(self):
        # method called when the manager stops

    def restart(self):
        # method called when the manager restarts
```

You can use this structure for anything you want, (even add an app to the loop).

To help you in your work a pyuv implementation of tornado is integrated and the powerful events module allows you to manage PUB/SUB events (or anything evented) inside your app. An EventEmitter is a threadsafe class to manage subscriber and publisher of events. It is internally used to broadcast process and manager events.

Watch stats

Stats of a process can be monitored continuously (there is a refresh interval of 0.1s to fetch CPU informations) using the following method:

```
manager.monitor(<nameorid>, <listener>)
```

Where <*nameorid*> is the name of the process template. In this case the statistics of all the the OS processes using this template will be emitted. Stats events are collected in the listener callback.

Callback signature: `callback(evtype, msg)`.

evtype is always "STATS" here and **msg** is a dict:

```
{
    "mem_info1: int,
    "mem_info2: int,
    "cpu": int,
    "mem": int,
    "ctime": int,
    "pid": int,
    "username": str,
    "nicce": int,
    "cmdline": str,
    "children": [{ stat dict, ... }]
}
```

To unmonitor the process in your app run:

```
manager.unmonitor(<nameorid>, <listener>)
```

Note: Internally a monitor subscribe you to an EventEmitter. A timer is running until there are subscribers to the process stats events.

Of course you can directly monitor a process using the internal pid:

```
process = manager.running[pid]
process.monitor(<listener>)
```

```
...

process.unmonitor(<listener>)
```

IO Events

Subscribe to stdout/stderr process streams

You can subscribe to stdout/stderr process streams and even write to stdin if you want.

To be able to receive the stdout/stderr streams in your application, you need to create a process with the *redirect_output* setting:

```
manager.add_process("nameofprocestemplate", cmd,
    redirect_output["stdout", "stderr"])
```

Note: Name of outputs can be anything, only the order counts. So if you want to name *stdout* as *a* just replace *stdout* by *a* in the declaration.

If you don't want to receive *stderr*, just omit it from the list.

If you want to redirect stderr to stdout just use the same name.

Then for example, to monitor the stdout output do:

```
process.monitor_io("stdout", somecallback)
```

Callback signature: `callback(evtype, msg)`.

And to unmonitor:

```
process.unmonitor_io("stdout", somecallback)
```

Note: To subscribe to all process streams replace the stream name by '.'' .

Write to STDIN

Writing to stdin is pretty easy. Just do:

```
process.write("somedata")
```

or to send multiple lines:

```
process.writelines(["line", "line"])
```

You can write lines from multiple publishers and multiple publishers can write at the same time. This method is threadsafe.

HTTP API

See the HTTP api description for more informations.

Tools

Gaffer proposes different tools (and more will come soon) to manage your processes without having to code. It can be used like supervisor, god, runit or other tools around. Speaking of runit a similar controlling will be available in 0.2 .

See the Command Line documentation for more informations.

1.2.3 CHANGES

2013/09/29 - version 0.5.1

- new intermediary release. See https://github.com/benoitc/gaffer/compare/0.4.4...0.5.0 for a detailled changelog.

2012/12/20 - version 0.4.4

- improve Events dispatching
- add support for multiple channel in a process
- add ping handler for monitoring
- some fixes in the http api
- fix `stop_processes` function

2012/11/02 - version 0.4.3

- process os environment now inherits from the gafferd environment
- fix autorestart feature: now handled asynchronously which allows us to still handle "stop command when a process fails"

2012/11/01 - version 0.4.2

- fix os_env option

2012/10/29 - version 0.4.0

- add environent variables support in the gafferd setting file.
- add a plugin system to easily extend Gafferd using HTML sites or gaffer applications in Python

2012/10/18 - version 0.3.1

- add environment variables substitution in the process command line and arguments.

2012/10/18 - version 0.3.0

- add the Gaffer command line tool: load, unload your procfile applications to gaffer, scale them up and down. Or just use it as a procfile manager just like foreman .
- add gafferctl `commands/watch` command to watch a node activity remotely.
- add priority feature: now processes can be launch in order
- add the possibility to manipulate groups of processes
- add the possibility to set the default endpoint in gafferd from the command line
- add `-v` and `--vv` options to gafferd to have a verbose output.
- add an eventsource client in the framework to manipulate gaffer streams.
- add `Manager.start_processes` method. Start all processes.
- add console_output application to the framework
- add new global Gaffer events to the manager: spawn, reap, stop_pid, exit.
- fix shutdown
- fix heartbeat

2012/10/15 - version 0.2.0

- add Webhooks: post to an url when a gaffer event is triggered
- add graceful shutdown. kill processes after a graceful time
- add `commands/load_process` command
- code refactoring: make the code simpler

2012/10/12 - version 0.1.0

Initial release

1.2.4 Command Line

Gaffer is a process management framework but also a set of command lines tools allowing you to manage on your machine or a cluster. All the command line tools are obviously using the framework.

Gaffer is an interface to the gaffer HTTP api and include support for loading/unloadin apps, scaling them up and down, ... It can also be used as a manager for Procfile-based applications similar to foreman but using the gaffer framework. It is running your application directly using a Procfile or export it to a gafferd configuration file or simply to a JSON file that you could send to gafferd using the HTTP api.

Gafferd is a server able to launch and manage processes. It can be controlled via the HTTP api. It is controlled by gafferctl and can be used to handle many processes.

Gaffer

The **gaffer** command line tool is an interface to the gaffer HTTP api and include support for loading/unloading Procfile applications, scaling them up and down, ...

It can also be used as a manager for Procfile-based applications similar to foreman but using the gaffer framework. It is running your application directly using a Procfile or export it to a gafferd configuration file or simply to a JSON file that you could send to gafferd using the HTTP api.

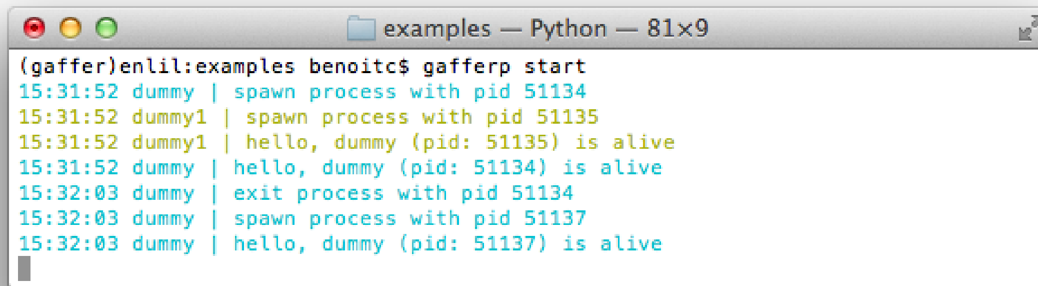
Example of use

For example using the following **Procfile**:

```
dummy: python -u dummy_basic.py
dummy1: python -u dummy_basic.py
```

You can launch all the programs in this procfile using the following command line:

```
$ gaffer start
```

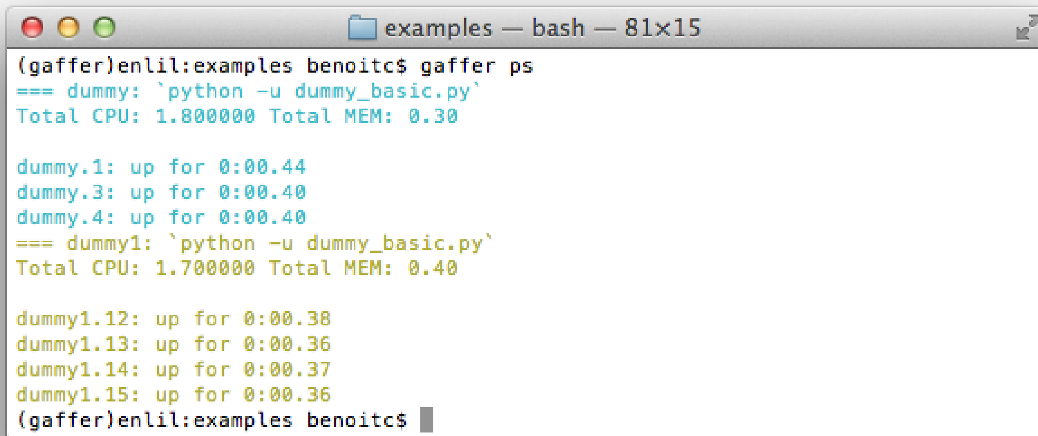


Or load them on a gaffer node:

```
$ gaffer load
```

and then scale them up and down:

```
$ gaffer scale dummy=3 dummy1+2
Scaling dummy processes... done, now running 3
Scaling dummy1 processes... done, now running 3
```



OPTIONS

-h –help show this help message and exit –version show version and exit -f procfile,–procfile procfile Specify an alternate Procfile to load -d root,–directory root Specify an alternate application root

This defaults to the directory containing the Procfile [default: .]

-e k=v,–env k=v Specify one or more .env files to load –endpoint endpoint gafferd node URL to connect

[default: http://127.0.0.1:5000]

SUBCOMMANDS

export [-c concurrency|–concurrency concurrency] [–format=format] [–out=filename] [<name>]

Export a Procfile

This command export a Procfile to a gafferd process settings format. It can be either a JSON that you could send to gafferd via the JSON API or an ini file that can be included to the gafferd configuration.

<format> ini or json –out=filename path of filename where the export will be saved

load [-c concurrency|–concurrency concurrency] [<name>] Load a Procfile application to gafferd

<name> is the name of the application recorded in gafferd. By default it will be the name of your project folder.You can use `.` to specify the current folder.

ps [<appname>] List your processes informations

<appname> he name of the application (session) of process recoreded in gafferd. By default it will be the name of your project folder.You can use `.` to specify the current folder.

run [-c] [<args>]... Run one-off commands using the same environment as your defined processes

-c concurrency Specify the number of each process type to run. The value passed in should be in the format process=num,process=num

--concurrency concurrency same as the -c option.

scale [<appname>] [process=value]... Scaling your process

Procfile applications can scale up or down instantly from the command line or API.

Scaling a process in an application is done using the scale command:

```
$ gaffer scale dummy=3
Scaling dummy processes... done, now running 3
```

Or both at once:

```
$ gaffer scale dummy=3 dummy1+2
Scaling dummy processes... done, now running 3
Scaling dummy1 processes... done, now running 3
```

start [-c concurrency|–concurrency concurrency]

Start a process type or all process types from the Procfile.

-c concurrency Specify the number of each process type to run. The value passed in should be in the format process=num,process=num

--concurrency concurrency same as the -c option.

unload [<name>] Unload a Procfile application from a gafferd node

Gafferd

Gafferd is a server able to launch and manage processes. It can be controlled via the HTTP api .

Usage

```
$ gafferd -h
usage: gafferd [-h] [-c CONFIG_FILE] [-p PLUGINS_DIR] [-v] [-vv] [--daemon]
               [--pidfile PIDFILE] [--bind BIND] [--certfile CERTFILE]
               [--keyfile KEYFILE] [--backlog BACKLOG]
               [config]

Run some watchers.

positional arguments:
  config                configuration file

optional arguments:
  -h, --help            show this help message and exit
  -c CONFIG_FILE, --config CONFIG_FILE
                        configuration file
  -p PLUGINS_DIR, --plugins-dir PLUGINS_DIR
                        default plugin dir
  -v                    verbose mode
  -vv                   like verbose mode but output stream too
  --daemon              Start gaffer in the background
  --pidfile PIDFILE
  --bind BIND           default HTTP binding
  --certfile CERTFILE   SSL certificate file for the default binding
```

```
  --keyfile KEYFILE     SSL key file for the default binding
  --backlog BACKLOG     default backlog
```

Config file example

```
[gaffer]
http_endpoints = public

[endpoint:public]
bind = 127.0.0.1:5000
;certfile=
;keyfile=

[webhooks]
;create = http://some/url
;proc.dummy.spawn = http://some/otherurl


[process:dummy]
cmd = ./dummy.py
;cwd = .
;uid =
;gid =
;detach = false
;shell = false
; flapping format: attempts=2, window=1., retry_in=7., max_retry=5
;flapping = 2, 1., 7., 5
numprocesses = 1
redirect_output = stdout, stderr
; redirect_input  = true
; graceful_timeout = 30

[process:echo]
cmd = ./echo.py
numprocesses = 1
redirect_output = stdout, stderr
redirect_input  = true
```

Plugins

Plugins are a way to enhance the basic gafferd functionality in a custom manner. Plugins allows you to load any gaffer application and site plugins. You can for example use the plugin system to add a simple UI to administrate gaffer using the HTTP interface.

A plugin has the following structure:

```
/pluginname
    _site/
    plugin/
        __init__.py
        ...
        ***.py
```

A plugin can be discovered by adding one ore more module that expose a class inheriting from `gaffer.Plugin`. Every plugin file should have a __all__ attribute containing the implemented plugin class. Ex:

```
from gaffer import Plugin

__all__ = ['DummyPlugin']

from .app import DummyApp


class DummyPlugin(Plugin):
    name = "dummy"
    version = "1.0"
    description = "test"

    def app(self, cfg):
        return DummyApp()
```

The dummy app here only print some info when started or stopped:

```
class DummyApp(object):

    def start(self, loop, manager):
        print("start dummy app")

    def stop(sef):
        print("stop dummy")

    def rester(self):
        print("restart dummy")
```

See the Overview for more infos. You can try it in the example folder:

```
$ cd examples
$ gafferd -c gaffer.ini -p plugins/
```

Install plugins Installing plugins can be done by placing the plugin in the plugin folder. The plugin folder is either set in the setting file using the **plugin_dir** in the gaffer section or using the `-p` option of the command line.

The default plugin dir is set to `~/.gafferd/plugins`.

Site plugins Plugins can have "sites" in them, any plugin that exists under the plugins directory with a _site directory, its content will be statically served when hitting `/_plugin/[plugin_name]/` url. Those can be added even after the process has started.

Installed plugins that do not contain any Python related content, will automatically be detected as site plugins, and their content will be moved under _site.

Mandatory Plugins If you rely on some plugins, you can define mandatory plugins using the `mandatory` attribute of a the plugin class, for example, here is a sample config:

```
class DummyPlugin(Plugin):
    ...
    mandatory = ['somedep']
```

gaffer_lookupd

Gaffer lookupd server to register gafferd nodes and access to their address.

$ gaffer_lookupd -h usage: gaffer_lookupd [–version] [-v] [–daemon] [–pidfile=PIDFILE]

[–bind=ADDRESS] [–backlog=BACKLOG] [–certfile=CERTFILE] [–keyfile=KEYFILE] [–cacert=CACERT]

Options

-h –help show this help message and exit –version show version and exit -v verbose mode –daemon Start gaffer in daemon mode –pidfile=PIDFILE –backlog=BACKLOG default backlog [default: 128] –bind=ADDRESS default HTTP binding [default: 0.0.0.0:5010] –certfile=CERTFILE SSL certificate file –keyfile=KEYFILE SSL key file –cacert=CACERT SSL CA certificate

1.2.5 HTTP api

an http API provided by the `gaffer.http_handler.HttpHandler`‘ gaffer application can be used to control gaffer via HTTP. To embed it in your app just initialize your manager with it:

```
manager = Manager(apps=[HttpHandler()])
```

The HttpHandler can be configured to accept multiple endpoinds and can be extended with new HTTP handlers. Internally we are using Tornado so you can either extend it with rules using pure totrnado handlers or wsgi apps.

Request Format and Responses

Gaffer supports **GET**, **POST**, **PUT**, **DELETE**, **OPTIONS** HTTP verbs.

All messages (except some streams) are JSON encoded. All messages sent to gaffers should be json encoded.

Gaffer supports cross-origin resource sharing (aka CORS).

HTTP endpoints

Main http endpoints are described in the description of the gafferctl commands in `gafferctl`:

Gafferctl is using extensively this HTTP api.

Output streams

The output streams can be fetched by doing:

```
GET /streams/<pid>/<nameofeed>
```

It accepts the following query parameters:

- **feed** : continuous, longpoll, eventsource
- **heartbeat**: true or seconds, send an empty line each sec (if true 60)

ex:

```
$ curl localhost:5000/streams/1/stderr?feed=continuous
STDERR 12
STDERR 13
STDERR 14
STDERR 15
STDERR 16
STDERR 17
```

```
STDERR 18
STDERR 19
STDERR 20
STDERR 21
STDERR 22
STDERR 23
STDERR 24
STDERR 25
STDERR 26
STDERR 27
STDERR 28
STDERR 29
STDERR 30
STDERR 31

$ curl localhost:5000/streams/1/stderr?feed=longpoll
STDERR 215

$ curl localhost:5000/streams/1/stderr?feed=eventsource
event: stderr
data: STDERR 20

event: stderr
data: STDERR 21

event: stderr
data: STDERR 22

$ curl localhost:5000/streams/1/stdout?feed=longpoll
STDOUTi 14
```

Write to STDIN

It is now possible to write to stdin via the HTTP api by sending:

```
POST to /streams/<pid>/ttin
```

Where <pid> is an internal process ide that you can retrieve by calling *GET /processses/<name>/_pids*

ex:

```
$ curl -XPOST -d $'ECHO\n' localhost:5000/streams/2/stdin
{"ok": true}

$ curl localhost:5000/streams/2/stdout?feed=longpoll
ECHO
```

Websocket stream for STDIN/STDOUT

It is now possible to get stin/stdout via a websocket. Writing to `ws://HOST:PORT/wstreams/<pid>` will send the data to stdin any information written on stdout will be then sent back to the websocket.

See the echo client/server example in the example folder:

```
$ python echo_client.py
Sent
Reeiving...
```

```
Received 'ECHO

'
(test)enlil:examples benoitc$ python echo_client.py
Sent
Reeiving...
Received 'ECHO
```

Note: unfortunately the echo_client script can only be launched with python 2.7 :/

Note: to redirect stderr to stdout just use the same name when you setting the redirect_output property on process creation.

1.2.6 Webhooks

1.2.7 Core gaffer framework

`manager` Module

The manager module is a core component of gaffer. A Manager is responsible of maintaining processes and allows you to interract with them.

Classes

class `gaffer.manager.`**`Manager`**(*loop=None*)

Bases: `object`

Manager - maintain process alive

A manager is responsible of maintaining process alive and manage actions on them:

- increase/decrease the number of processes / process template
- start/stop processes
- add/remove process templates to manage

The design is pretty simple. The manager is running on the default event loop and listening on events. Events are sent when a process exit or from any method call. The control of a manager can be extended by adding apps on startup. For example gaffer provides an application allowing you to control processes via HTTP.

Running an application is done like this:

```
# initialize the application with the default loop
loop = pyuv.Loop.default_loop()
m = Manager(loop=loop)

# start the application
m.start(apps=[HttpHandler])

.... # do smth

m.stop() # stop the controlller
m.run() # run the event loop
```

Note: The loop can be omitted if the first thing you do is launching a manager. The run function is here for convenience. You can of course just run *loop.run()* instead

Warning: The manager should be stopped the last one to prevent any lock in your application.

active

commit (*name*, *graceful_timeout=0*, *env=None*)
Like ‘‘scale(1) but the process won’t be kept alived at the end. It is also not handled uring scaling or reaping.

get (*name*)
get a job config

get_process (*pid*)
get an OS process by ID. A process is a `gaffer.Process` instance attached to a process state that you can use.

get_process_id ()
generate a process id

info (*name*)
get job’ infos

jobs (*sessionid=None*)

jobs_walk (*callback*, *sessionid=None*)

kill (*pid*, *sig*)
send a signal to a process

killall (*name*, *sig*)
send a signal to all processes of a job

list (*name=None*)

load (*config*, *sessionid=None*, *env=None*, *start=True*)
load a process config object.

Args:

- **config**: a `process.ProcessConfig` instance
- **sessionid**: Some processes only make sense in certain contexts. this flag instructs gaffer to maintain this process in the sessionid context. A context can be for example an application. If no session is specified the config will be attached to the `default` session.
- **env**: dict, None by default, if specified the config env variable will be updated with the env values.

manage (*name*)

monitor (*listener*, *name*)
get stats changes on a process template or id

pids (*name=None*)

reload (*name*, *sessionid=None*)
reload a process config. The number of processes is resetted to the one in settings and all current processes are killed

restart (*callback=None*)
restart all processes in the manager. This function is threadsafe

run()
Convenience function to use in place of *loop.run()* If the manager is not started it raises a *RuntimeError*.

Note: if you want to use separately the default loop for this thread then just use the start function and run the loop somewhere else.

scale(*name*, *n*)
Scale the number of processes in for a job. By using this function you can increase, decrease or set the number of processes in a template. Change is handled once the event loop is idling

n can be a positive or negative integer. It can also be a string containing the opetation to do. For example:

```
m.scale("sometemplate", 1) # increase of 1
m.scale("sometemplate", -1) # decrease of 1
m.scale("sometemplate", "+1") # increase of 1
m.scale("sometemplate", "-1") # decrease of 1
m.scale("sometemplate", "=1") # set the number of processess to 1
```

send(*pid*, *lines*, *stream=None*)
send some data to the process

sessions

start(*apps=[]*)
start the manager.

start_job(*name*)
Start a job from which the config have been previously loaded

stats(*name*)
return job stats

stop(*callback=None*)
stop the manager. This function is threadsafe

stop_job(*name*)
stop a jon. All processes of this job are stopped and won't be restarted by the manager

stop_process(*pid*)
stop a process

stopall(*name*)
stop all processes of a job. Processes are just exiting and will be restarted by the manager.

subscribe(*topic*)

unload(*name_or_process*, *sessionid=None*)
unload a process config.

unmonitor(*listener*, *name*)
get stats changes on a process template or id

unsubscribe(*topic*, *channel*)

update(*config*, *sessionid=None*, *env=None*, *start=False*)
update a process config. All processes are killed

wakeup()

walk(*callback*, *name=None*)

process Module

Gaffer events

Many events happend in gaffer.

Manager events

Manager events have the following format:

```
{
  "event": "<nameofevent">>,
  "name": "<templatename>"
}
```

- **create**: a process template is created
- **start**: a process template start to launch OS processes
- **stop**: all OS processes of a process template are stopped
- **restart**: all processes of a process template are restarted
- **update**: a process template is updated
- **delete**: a process template is deleted
- **spawn**: a new process is spawned
- **reap**: a process is reaped
- **exit**: a process exited
- **stop_pid**: a process has been stopped

Processes events

All processes' events are prefixed by `proc.<name>` to make the pattern matching easier, where `<name>` is the name of the process template

Events are:

- **proc.<name>.start** : the template <name> start to spawn processes
- **proc.<name>.spawn** : one OS process using the process <name> template is spawned. Message is:

```
{
  "event": "proc.<name>.spawn">>,
  "name": "<name>",
  "detach": false,
  "pid": int
}
```

  Note: pid is the internal pid

- **proc.<name>.exit**: one OS process of the <name> template has exited. Message is:

```
{
  "event": "proc.<name>.exit">>,
  "name": "<name>",
  "pid": int,
  "exit_code": int,
  "term_signal": int
}
```

- **proc.<name>.stop**: all OS processes in the template <name> are stopped.
- **proc.<name>.stop_pid**: One OS process of the template <name> is stopped. Message is:

```
{
  "event": "proc.<name>.stop_pid">>,
  "name": "<name>",
  "pid": int
}
```

- **proc.<name>.stop_pid**: One OS process of the template <name> is reapped. Message is:

```
{
  "event": "proc.<name>.reap">>,
  "name": "<name>",
  "pid": int
}
```

The `events` Module

This module offeres a common way to susbscribe and emit events. All events in gaffer are using.

Example of usage

```
event = EventEmitter()

# subscribe to all events with the pattern a.*
event.subscribe("a", subscriber)

# subscribe to all events "a.b"
event.subscribe("a.b", subscriber2)

# subscribe to all events (wildcard)
event.subscribe(".", subscriber3)

# publish an event
event.publish("a.b", arg, namedarg=val)
```

In this example all subscribers will be notified of the event. A subscriber is just a callable *(event, *args, **kwargs)*

Classes

class `gaffer.events.`**`EventEmitter`**(*loop*, *max_size=10000*)

Bases: `object`

Many events happend in gaffer. For example a process will emist the events "start", "stop", "exit".

This object offer a common interface to all events emitters

close()
close the event

This function clear the list of listeners and stop all idle callback

publish(*evtype*, **args*, ***kwargs*)
emit an event **evtype**

The event will be emitted asynchronously so we don't block here

subscribe(*evtype*, *listener*, *once=False*)
subcribe to an event

subscribe_once(*evtype*, *listener*)
subscribe to event once. Once the evennt is triggered we remove ourself from the list of listenerrs

unsubscribe(*evtype*, *listener*, *once=False*)
unsubscribe from an event

unsubscribe_all(*events=[]*)
unsubscribe all listeners from a list of events

unsubscribe_once(*evtype*, *listener*)

Webhooks

`procfile` Module

module to parse and manage a Procfile

class `gaffer.procfile.`**Procfile**(*procfile*, *root=None*, *envs=None*)
Bases: `object`

Procfile object to parse a procfile and a list of given environnment files.

as_configparser(*concurrency_settings=None*)
return a ConfigParser object. It can be used to generate a gafferd setting file or a configuration file that can be included.

as_dict(*name*, *concurrency_settings=None*)
return a procfile line as a JSON object usable with the command `gafferctl load`.

get_appname()

parse(*procfile*)
main function to parse a procfile. It returns a dict

parse_cmd(*v*)

processes()
iterator over the configuration

`gaffer.procfile.`**get_env**(*envs=[]*)

`pidfile` Module

class `gaffer.pidfile.`**Pidfile**(*fname*)
Bases: `object`

Manage a PID file. If a specific name is provided it and '"%s.oldpid" % name' will be used. Otherwise we create a temp file using os.mkstemp.

create(*pid*)

rename(*path*)

unlink()
delete pidfile

validate()
Validate pidfile and make it stale if needed

util Module

gaffer.util.**bind_sockets**(*addr*, *backlog=128*, *allows_unix_socket=False*)

gaffer.util.**bytes2human**(*n*)
Translates bytes into a human repr.

gaffer.util.**bytestring**(*s*)

gaffer.util.**check_gid**(*val*)
Return a gid, given a group value

If the group value is unknown, raises a ValueError.

gaffer.util.**check_uid**(*val*)
Return an uid, given a user value. If the value is an integer, make sure it's an existing uid.

If the user value is unknown, raises a ValueError.

gaffer.util.**daemonize**()
Standard daemonization of a process.

gaffer.util.**from_nanotime**(*n*)
convert from nanotime to seconds

gaffer.util.**getcwd**()
Returns current path, try to use PWD env first

gaffer.util.**hostname**()

gaffer.util.**is_ipv6**(*addr*)

gaffer.util.**is_ssl**(*url*)

gaffer.util.**nanotime**(*s=None*)
convert seconds to nanoseconds. If s is None, current time is returned

gaffer.util.**ord_**(*c*)

gaffer.util.**parse_address**(*netloc*, *default_port=8000*)

gaffer.util.**parse_job_name**(*name*, *default='default'*)

gaffer.util.**parse_signal_value**(*sig*)

gaffer.util.**parse_ssl_options**(*client_options*)

gaffer.util.**setproctitle_**(*title*)

gaffer.util.**substitute_env**(*s*, *env*)

tornado_pyuv Module

class gaffer.tornado_pyuv.**IOLoop**(*impl=None*, *_loop=None*)
Bases: object

ERROR = 24

NONE = 0

READ = 1

WRITE = 4

add_callback(*callback*)

add_handler(*fd*, *handler*, *events*)

add_timeout(*deadline*, *callback*)

close()

handle_callback_exception(*callback*)
This method is called whenever a callback run by the IOLoop throws an exception.

By default simply logs the exception as an error. Subclasses may override this method to customize reporting of exceptions.

The exception itself is not passed explicitly, but is available in sys.exc_info.

static **initialized**()
Returns true if the singleton instance has been created.

install()
Installs this IOLoop object as the singleton instance.

This is normally not necessary as *instance()* will create an IOLoop on demand, but you may want to call *install* to use a custom subclass of IOLoop.

static **instance**()

log_stack(*signal*, *frame*)

remove_handler(*fd*)

remove_timeout(*timeout*)

running()
Returns true if this IOLoop is currently running.

set_blocking_log_threshold(*seconds*)

set_blocking_signal_threshold(*seconds*, *action*)

start()

stop()

update_handler(*fd*, *events*)

class gaffer.tornado_pyuv.**PeriodicCallback**(*callback*, *callback_time*, *io_loop=None*)
Bases: object

start()

stop()

class gaffer.tornado_pyuv.**Waker**(*loop*)
Bases: object

wake()

gaffer.tornado_pyuv.**install**()

1.2.8 `httpclient` Module

1.2.9 Gaffer applications

Gaffer applications are applications that are started by the manager. A gaffer application can be used to interract with the manager or listening on events.

An application is a class with the following structure:

```
class Myapplication(object):

    def __init__(self):
        # do inti

    def start(self, loop, manager):
        # this method is call by the manager to start the
        application

    def stop(self):
        # method called when the manager stop

    def restart(self):
        # methhod called when the manager restart
```

Following applications are provided by gaffer:

`http_handler` Module

`console_output` Module

module to return all streams from the managed processes to the console. This application is subscribing to the manager to know when a process is created or killed and display the information. When an OS process is spawned it then subscribe to its streams if any are redirected and print the output on the console. This module is used by Gaffer .

Note: if colorize is set to true, each templates will have a different colour

class gaffer.console_output.**Color**

Bases: object

wrapper around colorama to ease the output creation. Don't use it directly, instead, use the `colored(name_of_color, lines)` to return the colored ouput.

Colors are: cyan, yellow, green, magenta, red, blue, intense_cyan, intense_yellow, intense_green, intense_magenta, intense_red, intense_blue.

lines can be a list or a string.

output(*color_name*, *lines*)

class gaffer.console_output.**ConsoleOutput**(*colorize=True*, *output_streams=True*, *actions=None*)

Bases: object

The application that need to be added to the gaffer manager

DEFAULT_ACTIONS = ['start', 'stop', 'spawn', 'reap', 'exit', 'stop_pid']

restart()

start(*loop*, *manager*)

stop()

gaffer.console_output.**status_bar**(*s*)

sig_handler Module

class gaffer.sig_handler.**BaseSigHandler**

Bases: object

A simple gaffer application to handle signals

QUIT_SIGNALS = (3, 15, 2)

handle_quit(*handle*, *signum*)

handle_reload(*handle*, *signum*)

restart()

start(*loop*)

stop()

class gaffer.sig_handler.**SigHandler**

Bases: *gaffer.sig_handler.BaseSigHandler*

A simple gaffer application to handle signals

handle_quit(*handle*, **args*)

handle_reload(*handle*, **args*)

start(*loop*, *manager*)

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

gaffer.console_output, 27
gaffer.events, 22
gaffer.manager, 19
gaffer.pidfile, 24
gaffer.procfile, 24
gaffer.sig_handler, 28
gaffer.tornado_pyuv, 26
gaffer.util, 25

A

active (gaffer.manager.Manager attribute), 20
add_callback() (gaffer.tornado_pyuv.IOLoop method), 26
add_handler() (gaffer.tornado_pyuv.IOLoop method), 26
add_timeout() (gaffer.tornado_pyuv.IOLoop method), 26
as_configparser() (gaffer.procfile.Procfile method), 24
as_dict() (gaffer.procfile.Procfile method), 24

B

BaseSigHandler (class in gaffer.sig_handler), 28
bind_sockets() (in module gaffer.util), 25
bytes2human() (in module gaffer.util), 25
bytestring() (in module gaffer.util), 25

C

check_gid() (in module gaffer.util), 25
check_uid() (in module gaffer.util), 25
close() (gaffer.events.EventEmitter method), 23
close() (gaffer.tornado_pyuv.IOLoop method), 26
Color (class in gaffer.console_output), 27
commit() (gaffer.manager.Manager method), 20
ConsoleOutput (class in gaffer.console_output), 27
create() (gaffer.pidfile.Pidfile method), 24

D

daemonize() (in module gaffer.util), 25
DEFAULT_ACTIONS (gaffer.console_output.ConsoleOutput attribute), 28

E

ERROR (gaffer.tornado_pyuv.IOLoop attribute), 26
EventEmitter (class in gaffer.events), 23

F

from_nanotime() (in module gaffer.util), 25

G

gaffer.console_output (module), 27
gaffer.events (module), 22
gaffer.manager (module), 19
gaffer.pidfile (module), 24
gaffer.procfile (module), 24
gaffer.sig_handler (module), 28
gaffer.tornado_pyuv (module), 26
gaffer.util (module), 25
get() (gaffer.manager.Manager method), 20
get_appname() (gaffer.procfile.Procfile method), 24
get_env() (in module gaffer.procfile), 24
get_process() (gaffer.manager.Manager method), 20
get_process_id() (gaffer.manager.Manager method), 20
getcwd() (in module gaffer.util), 25

H

handle_callback_exception() (gaffer.tornado_pyuv.IOLoop method), 26
handle_quit() (gaffer.sig_handler.BaseSigHandler method), 28
handle_quit() (gaffer.sig_handler.SigHandler method), 28
handle_reload() (gaffer.sig_handler.BaseSigHandler method), 28
handle_reload() (gaffer.sig_handler.SigHandler method), 28
hostname() (in module gaffer.util), 25

I

info() (gaffer.manager.Manager method), 20
initialized() (gaffer.tornado_pyuv.IOLoop static method), 26
install() (gaffer.tornado_pyuv.IOLoop method), 26
install() (in module gaffer.tornado_pyuv), 27
instance() (gaffer.tornado_pyuv.IOLoop static method), 26
IOLoop (class in gaffer.tornado_pyuv), 26
is_ipv6() (in module gaffer.util), 25
is_ssl() (in module gaffer.util), 25

J

jobs() (gaffer.manager.Manager method), 20
jobs_walk() (gaffer.manager.Manager method), 20

K

kill() (gaffer.manager.Manager method), 20
killall() (gaffer.manager.Manager method), 20

L

list() (gaffer.manager.Manager method), 20
load() (gaffer.manager.Manager method), 20
log_stack() (gaffer.tornado_pyuv.IOLoop method), 26

M

manage() (gaffer.manager.Manager method), 20
Manager (class in gaffer.manager), 19
monitor() (gaffer.manager.Manager method), 20

N

nanotime() (in module gaffer.util), 25
NONE (gaffer.tornado_pyuv.IOLoop attribute), 26

O

ord_() (in module gaffer.util), 25
output() (gaffer.console_output.Color method), 27

P

parse() (gaffer.procfile.Procfile method), 24
parse_address() (in module gaffer.util), 25
parse_cmd() (gaffer.procfile.Procfile method), 24
parse_job_name() (in module gaffer.util), 25
parse_signal_value() (in module gaffer.util), 25
parse_ssl_options() (in module gaffer.util), 25
PeriodicCallback (class in gaffer.tornado_pyuv), 26
Pidfile (class in gaffer.pidfile), 24
pids() (gaffer.manager.Manager method), 20
processes() (gaffer.procfile.Procfile method), 24
Procfile (class in gaffer.procfile), 24
publish() (gaffer.events.EventEmitter method), 24

Q

QUIT_SIGNALS (gaffer.sig_handler.BaseSigHandler attribute), 28

R

READ (gaffer.tornado_pyuv.IOLoop attribute), 26
reload() (gaffer.manager.Manager method), 20
remove_handler() (gaffer.tornado_pyuv.IOLoop method), 26
remove_timeout() (gaffer.tornado_pyuv.IOLoop method), 26
rename() (gaffer.pidfile.Pidfile method), 25
restart() (gaffer.console_output.ConsoleOutput method), 28
restart() (gaffer.manager.Manager method), 20
restart() (gaffer.sig_handler.BaseSigHandler method), 28
run() (gaffer.manager.Manager method), 20
running() (gaffer.tornado_pyuv.IOLoop method), 26

S

scale() (gaffer.manager.Manager method), 21
send() (gaffer.manager.Manager method), 21
sessions (gaffer.manager.Manager attribute), 21
set_blocking_log_threshold() (gaffer.tornado_pyuv.IOLoop method), 26
set_blocking_signal_threshold() (gaffer.tornado_pyuv.IOLoop method), 26
setproctitle_() (in module gaffer.util), 25
SigHandler (class in gaffer.sig_handler), 28
start() (gaffer.console_output.ConsoleOutput method), 28
start() (gaffer.manager.Manager method), 21
start() (gaffer.sig_handler.BaseSigHandler method), 28
start() (gaffer.sig_handler.SigHandler method), 28
start() (gaffer.tornado_pyuv.IOLoop method), 26
start() (gaffer.tornado_pyuv.PeriodicCallback method), 26
start_job() (gaffer.manager.Manager method), 21
stats() (gaffer.manager.Manager method), 21
status_bar() (in module gaffer.console_output), 28
stop() (gaffer.console_output.ConsoleOutput method), 28
stop() (gaffer.manager.Manager method), 21
stop() (gaffer.sig_handler.BaseSigHandler method), 28
stop() (gaffer.tornado_pyuv.IOLoop method), 26
stop() (gaffer.tornado_pyuv.PeriodicCallback method), 26
stop_job() (gaffer.manager.Manager method), 21
stop_process() (gaffer.manager.Manager method), 21
stopall() (gaffer.manager.Manager method), 21
subscribe() (gaffer.events.EventEmitter method), 24
subscribe() (gaffer.manager.Manager method), 21
subscribe_once() (gaffer.events.EventEmitter method), 24
substitute_env() (in module gaffer.util), 25

U

unlink() (gaffer.pidfile.Pidfile method), 25
unload() (gaffer.manager.Manager method), 21
unmonitor() (gaffer.manager.Manager method), 21
unsubscribe() (gaffer.events.EventEmitter method), 24
unsubscribe() (gaffer.manager.Manager method), 21
unsubscribe_all() (gaffer.events.EventEmitter method), 24
unsubscribe_once() (gaffer.events.EventEmitter method), 24
update() (gaffer.manager.Manager method), 21
update_handler() (gaffer.tornado_pyuv.IOLoop method), 26

V

validate() (gaffer.pidfile.Pidfile method), 25

W

wake() (gaffer.tornado_pyuv.Waker method), 26
Waker (class in gaffer.tornado_pyuv), 26
wakeup() (gaffer.manager.Manager method), 21
walk() (gaffer.manager.Manager method), 21
WRITE (gaffer.tornado_pyuv.IOLoop attribute), 26