gaffer Documentation

Release

Author

December 20, 2012

CONTENTS

CHAPTER

ONE

GAFFER

Application deployement, monitoring and supervision made simple.

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

1.1 Features

- Framework to manage and interact your processes
- Fully evented. Use the libuv event loop using the pyuv library
- Server and *Command Line* tools to manage your processes
- Procfile applications support (see *Gaffer*)
- HTTP Api (multiple binding, unix sockets & HTTPS supported)
- Flapping: handle cases where your processes crash too much
- **Possibility to interact with STDIO:**
  - websocket stream to write to stdin and receive from stdout (muliple clients can read and write at the same time)
  - subscribe on stdout/stderr feed via longpolling, continuous stream, eventsource or websockets
  - write your own client/server using the framework
- Subscribe to process statistics per process or process templates and get them in quasi RT.
- Easily extensible: add your own endpoint, create your client, embed gaffer in your application, ...
- Compatible with python 2.6x, 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/plugun
```

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*.

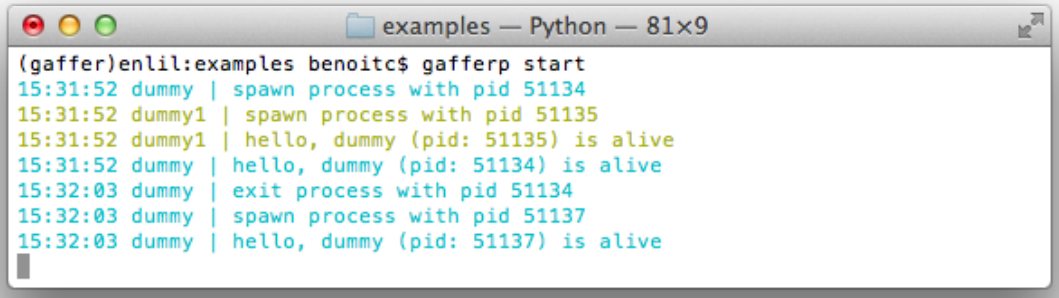
Example of use

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
```

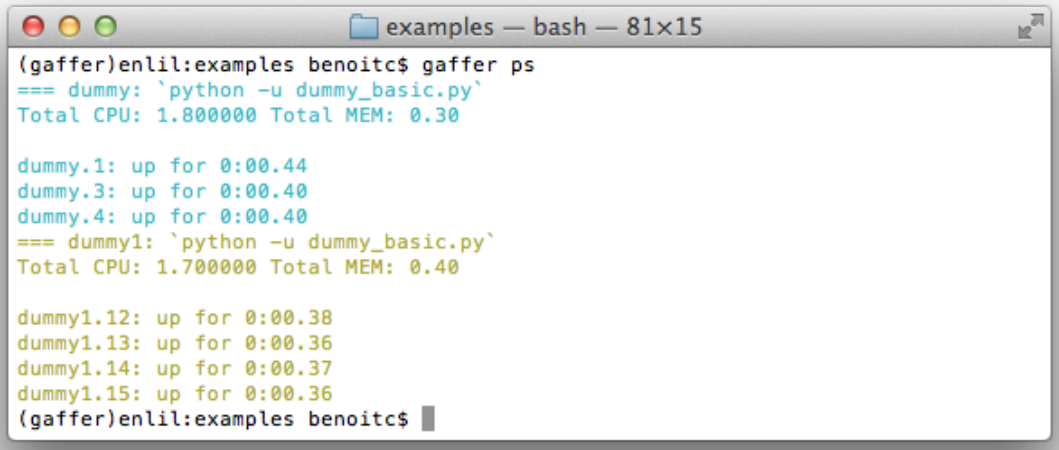
All processes in the Procfile will be then loaded to gafferd and started.

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
```

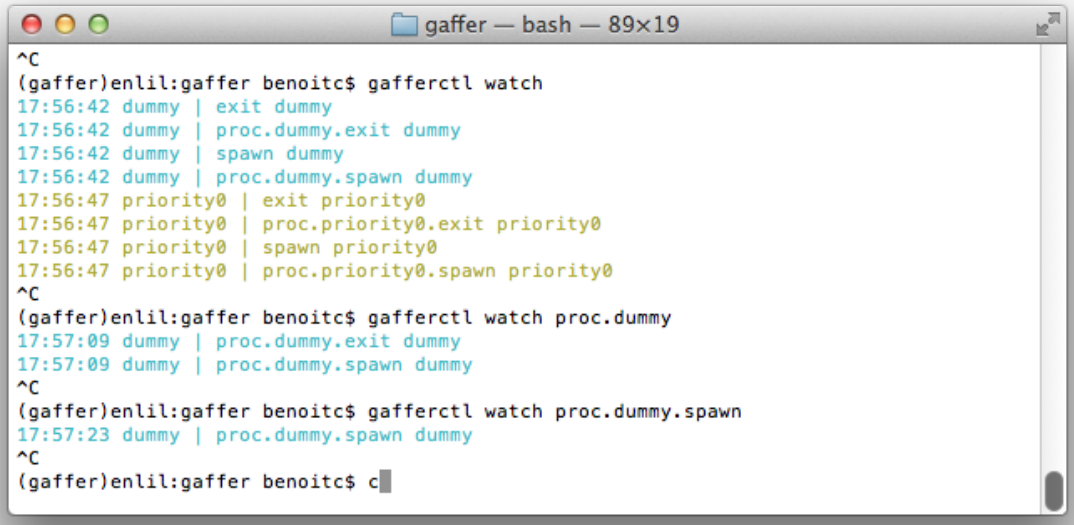
And stop it using the stop command.

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 ca 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 wich make it pretty performant to handle multiple process and their control.

At the lowest level you will find the manager. 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

A process template describe 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 of this templates are:

- **name**: name of the process
- **cmd**: program command, string)
- **args**: the arguments for the command to run. Can be a list or a string. If **args** is a string, it's splitted 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 he manager to react on different events. A applicaton can fetch informations from the manager and interract with him.

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):
        # do inti

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

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

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

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 a powerfull *events* modules will 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 processes and manager events.

Watch stats

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

```
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 monitor directly to a process using the internal pid:

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

...

process.unmonitor(<listener>)
```

IO Events

Subscribe to stdout/stderr process stream

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

To be able to receive the stdour/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 count 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 in the list. Alos 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 publisher 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 process 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

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 *Watch changes in gaffer* 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 *Load a process from a file* 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 yout to manage on your machine or a cluster. All the command line tools are obviously using the framework.

gaffer'is an interface to the :doc:'gaffer HTTP api and inclusde 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.

The tool *Gafferctl* allows you to control a local or remote gafferd node via the HTTP API. You can show processes informations, add new processes, changes their configureation, get changes on the nodes in rt ....

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*.

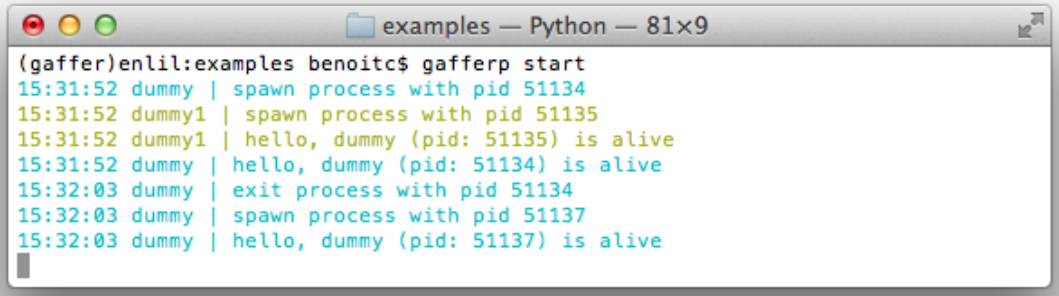
Example of use

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
```

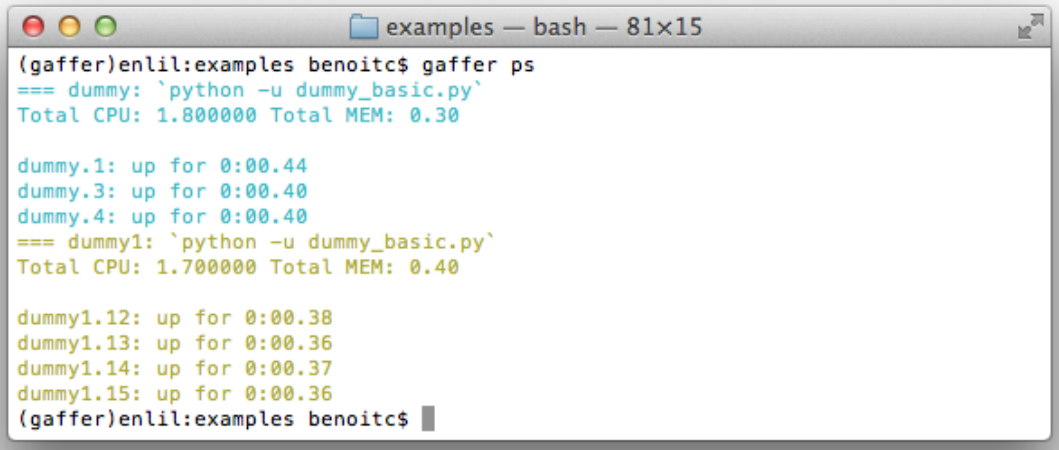


Or load them on a gaffer node:

```
$ 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
```



gaffer commands

- **start**: *Start a process*
- **run**: *Run one-off command*
- **export**: *Export a Procfile*
- **load**: *Load a Procfile application to gafferd*
- **unload**: *Unload a Procfile application to gafferd*
- **scale**: *Scaling your process*
- **ps**: *List your process informations*

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.

Command Line

```
$ gaffer export [ini|json] [filename]
```

Load a Procfile application to gafferd This command allows you to load your Procfile application in gafferd.

Command line

$ gaffer load [name] [url]

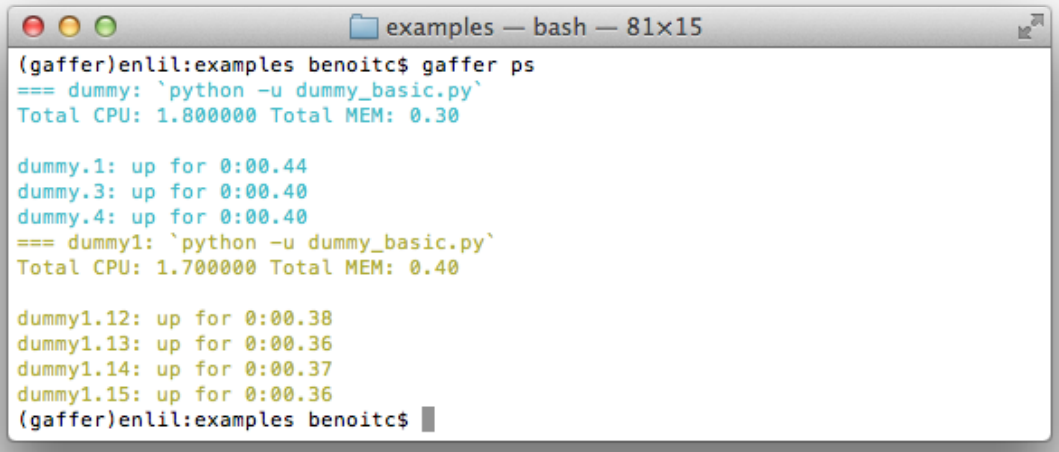
Arguments *name* is the name of the group of process recoreded in gafferd. By default it will be the name of your project folder.You can use `.` to specify the current folder.

uri is the url to connect to a gaffer node. By default 'http://127.0.0.1:5000'

Options **–endpoint**

Gaffer node URL to connect.

List your process informations Ps allows you to retrieve some process informations



Command line

```
$ gaffer ps [group]
```

Args *group* is the name of the group of process recoreded in gafferd. By default it will be the name of your project folder.You can use `.` to specify the current folder.

name is the name of one process

Run one-off command gaffer run is used to run one-off commands using the same environment as your defined processes.

Command line:

```
$ gaffer run /some/script
```

Options **-c**, **–concurrency**:

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

–env Specify one or more .env files to load

-f, –procfile: Specify an alternate Procfile to load

-d, **–directory**:

Specify an alternate application root. This defaults to the directory containing the Procfile

Scaling your process Procfile applications can scal 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
```

Command line

```
$ gaffer scale [group] process[=|-|+]3
```

Options –endpoint

Gaffer node URL to connect.

Operations supported are +,-,=

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

Command line

```
$ gaffer start [name]
```

Gaffer will run your application directly from the command line.

If no additional parameters are passed, gaffer run one instance of each type of process defined in your Procfile.

Options -c, **–concurrency**:

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

–env Specify one or more .env files to load

-f, –procfile: Specify an alternate Procfile to load

-d, **–directory**:

Specify an alternate application root. This defaults to the directory containing the Procfile

Unload a Procfile application to gafferd This command allows you to unload your Procfile application in gafferd.

Command line

$ gaffer unload [name] [url]

Arguments *name* is the name of the group of process recoreded in gafferd. By default it will be the name of your project folder.You can use `.` to specify the current folder.

uri is the url to connect to a gaffer node. By default 'http://127.0.0.1:5000'

Options –endpoint

Gaffer node URL to connect.

Command line usage

```
$ gaffer
usage: gaffer [options] command [args]

manage Procfiles applications.

optional arguments:
  -h, --help            show this help message and exit
  -c CONCURRENCY, --concurrency CONCURRENCY
                        Specify the number of each process type to run. The
                        value passed in should be in the format
                        process=num,process=num
  -e ENVS [ENVS ...], --env ENVS [ENVS ...]
                        Specify one or more .env files to load
  -f FILE, --procfile FILE
                        Specify an alternate Procfile to load
  -d ROOT, --directory ROOT
                        Specify an alternate application root. This defaults
                        to the directory containing the Procfile
  --endpoint ENDPOINT   Gaffer node URL to connect
  --version             show program's version number and exit

Commands:
---------

    start   Start a process
    run     Run one-off command
    export  Export a Procfile
    load    Load a Procfile application to gafferd
    unload  Unload a Procfile application to gafferd
    scale   Scaling your process
    ps      List your process informations
    help    Get help on a command
```

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']
```

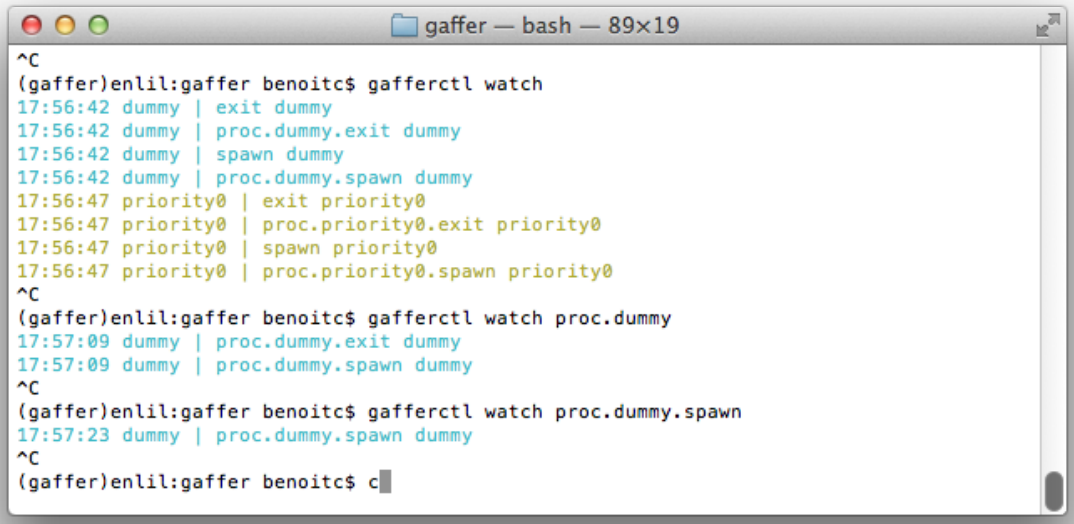
Gafferctl

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

```
$ gafferctl processes
```

gafferctl is an HTTP client able to connect to a UNIX pipe or a tcp connection and connect to a gaffer node. It is using the httpclient module to do it.

You can create your own client either by using the client API provided in the httpclient module or by reading the doc here and passing your own message to the gaffer node. All messages are encoded in JSON.



Usage

```
$ gafferctl help
usage: gafferctl [--version] [--connect=<endpoint>]
                 [--certfile] [--keyfile]
                 [--help]
                 <command> [<args>]

Commands:
    add             Increment the number of OS processes
    add_process     Add a process to monitor
    del_process     Get a process description
    get_process     Fetch a process template
```

```
help            Get help on a command
kill            Send a signal to a process
load_process    Load a process from a file
numprocesses    Number of processes that should be launched
pids            Get launched process ids for a process template
processes       Add a process to monitor
running         Number of running processes for this process description
start           Start a process
status          Return the status of a process
stop            Stop a process
sub             Decrement the number of OS processes
update_process  Update a process description
```

gafferctl commands

- **status**: *Return the status of a process*
- **processes**: *Add a process to monitor*
- **sub**: *Decrement the number of OS processes*
- **add_process**: *Add a process to monitor*
- **get_process**: *Fetch a process template*
- **stop**: *Stop a process*
- **running**: *Number of running processes for this process description*
- **load_process**: *Load a process from a file*
- **watch**: *Watch changes in gaffer*
- **start**: *Start a process*
- **add**: *Increment the number of OS processes*
- **update_process**: *Update a process description*
- **kill**: *Send a signal to a process*
- **numprocesses**: *Number of processes that should be launched*
- **del_process**: *Get a process description*
- **pids**: *Get launched process ids for a process template*

Increment the number of OS processes This command dynamically increase the number of OS processes for this process description to monitor in gafferd.

HTTP Message:

```
HTTP/1.1 POST /processes/<name>/_add/<inc>
```

The response return {"ok": true} with an http status 200 if everything is ok.

Properties:

- **name**: name of the process
- **inc**: The number of new OS processes to start

Command line:

```
gafferctl add name inc
```

Options

- <name>: name of the process to create
- <inc>: The number of new OS processes to start

Add a process to monitor This command dynamically add a process to monitor in gafferd.

HTTP Message:

```
HTTP/1.1 POST /processes
Content-Type: application/json
Accept: application/json

{
    "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
}
```

The response return {"ok": true} with an http status 200 if everything is ok.

It return a 409 error in case of a conflict (a process with this name has already been created.

Properties:

- **name**: name of the process
- **cmd**: program command, string)
- **args**: the arguments for the command to run. Can be a list or a string. If **args** is a string, it's splitted 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

Command line:

```
gafferctl add_process [--start] name cmd
```

Options

- <name>: name of the process to create
- <cmd>: full command line to execute in a process
- –start: start the watcher immediately

Get a process description This command stop a process and remove it from the monitored process.

HTTP Message:

```
HTTP/1.1 DELETE /processes/<name>
```

The response return {"ok": true} with an http status 200 if everything is ok.

Properties:

- **name**: name of the process

Command line:

```
gafferctl del_process name
```

Options

- <name>: name of the process to remove

Fetch a process template This command stop a process and remove it from the monitored process.

HTTP Message:

```
HTTP/1.1 GET /processes/<name>
```

The response return:

```
{
    "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
}
```

with an http status 200 if everything is ok.

Properties:

- **name**: name of the process
- **cmd**: program command, string)
- **args**: the arguments for the command to run. Can be a list or a string. If **args** is a string, it's splitted 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

Command line:

```
gafferctl get_process name
```

Options

- <name>: name of the process details to fetch

Send a signal to a process This command send any signal to a process by name or id.

HTTP Message

```
HTTP/1.1 POST /processes/<name_or_id>/_signal/<signum>
```

The response *{"ok": True}* if everything was ok.

Command line:

```
gafferctl kill <name_or_id> <signum>
```

Options

- <name_or_id>: name or id of the process
- <signum>: number or name, POSIX signal number (man signal or kill for more information):

```
No    Name         Default Action       Description
1     SIGHUP       terminate process    terminal line hangup
2     SIGINT       terminate process    interrupt program
3     SIGQUIT      create core image    quit program
4     SIGILL       create core image    illegal instruction
5     SIGTRAP      create core image    trace trap
6     SIGABRT      create core image    abort program (formerly SIGIOT)
7     SIGEMT       create core image    emulate instruction executed
8     SIGFPE       create core image    floating-point exception
```

```
9     SIGKILL      terminate process    kill program
10    SIGBUS       create core image    bus error
11    SIGSEGV      create core image    segmentation violation
12    SIGSYS       create core image    non-existent system call invoked
13    SIGPIPE      terminate process    write on a pipe with no reader
14    SIGALRM      terminate process    real-time timer expired
15    SIGTERM      terminate process    software termination signal
16    SIGURG       discard signal       urgent condition present on socket
17    SIGSTOP      stop process         stop (cannot be caught or ignored)
18    SIGTSTP      stop process         stop signal generated from keyboard
19    SIGCONT      discard signal       continue after stop
20    SIGCHLD      discard signal       child status has changed
21    SIGTTIN      stop process         background read attempted from control terminal
22    SIGTTOU      stop process         background write attempted to control terminal
23    SIGIO        discard signal       I/O is possible on a descriptor (see fcntl(2))
24    SIGXCPU      terminate process    cpu time limit exceeded (see setrlimit(2))
25    SIGXFSZ      terminate process    file size limit exceeded (see setrlimit(2))
26    SIGVTALRM    terminate process    virtual time alarm (see setitimer(2))
27    SIGPROF      terminate process    profiling timer alarm (see setitimer(2))
28    SIGWINCH     discard signal       Window size change
29    SIGINFO      discard signal       status request from keyboard
30    SIGUSR1      terminate process    User defined signal 1
31    SIGUSR2      terminate process    User defined signal 2
```

Load a process from a file Like the command `add`, his command dynamically add a process to monitor in gafferd. Informations are gathered from a file or stdin if the name of file is `-`. The file sent is a json file that have the same format described for the HTTP message.

HTTP Message:

```
HTTP/1.1 POST /processes
Content-Type: application/json
Accept: application/json

{
    "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
}
```

The response return {"ok": true} with an http status 200 if everything is ok.

It return a 409 error in case of a conflict (a process with this name has already been created.

Properties:

- **name**: name of the process
- **cmd**: program command, string)

- **args**: the arguments for the command to run. Can be a list or a string. If **args** is a string, it's splitted 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

Command line:

```
gafferctl load_process [--start] <file>
```

Options

- <name>: name of the process to create
- <file>: path to a json file or stdin –
- –start: start the watcher immediately

Example of usage:

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

Number of processes that should be launched This command return the number of processes that should be launched

HTTP Message:

```
HTTP/1.1 GET /status/<name>
```

The response return:

```
{
    "active": true,
    "running": 1,
    "numprocesses": 1
}
```

with an http status 200 if everything is ok.

Properties:

- **name**: name of the process

Command line:

```
gafferctl numprocesses name
```

Options

- <name>: name of the process to start

Get launched process ids for a process template This command return the list of launched process ids for a process template. Process ids are internals ids (for some reason we don't expose the system process ids)

HTTP Message:

```
HTTP/1.1 GET /processes/<name>/_pids
```

The response return:

```
{
    "ok": true,
    "pids": [1],
}
```

with an http status 200 if everything is ok.

Properties:

- **name**: name of the process

Command line:

```
gafferctl pids name
```

Options

- <name>: name of the process to start

Add a process to monitor This command dynamically add a process to monitor in gafferd.

HTTP Message:

```
HTTP/1.1 GET /processes?running=true
```

The response return a list of processes. If running=true it will return the list of running processes by pids (pids are internal process ids).

Command line:

```
gafferctl processes [--running]
```

Options

- <name>: name of the process to create
- –running: return the list of process by pid

Number of running processes for this process description This command return the number of processes that are currently running.

HTTP Message:

```
HTTP/1.1 GET /status/<name>
```

The response return:

```
{
    "active": true,
    "running": 1,
    "numprocesses": 1
}
```

with an http status 200 if everything is ok.

Properties:

- **name**: name of the process

Command line:

```
gafferctl running name
```

Options

- <name>: name of the process to start

Start a process This command dynamically start a process.

HTTP Message:

```
HTTP/1.1 POST /processes/<name>/_start
```

The response return {“ok”: true} with an http status 200 if everything is ok.

Properties:

- **name**: name of the process

Command line:

```
gafferctl start name
```

Options

- <name>: name of the process to start

Return the status of a process This command dynamically add a process to monitor in gafferd.

HTTP Message:

```
HTTP/1.1 GET /status/name
Content-Type: application/json
```

The response return:

```
{
    "active": true,
    "running": 1,
    "numprocesses": 1
}
```

with an http status 200 if everything is ok.

Properties:

- **name**: name of the process

Command line:

```
gafferctl status name
```

Options

- <name>: name of the process to create

Stop a process This command dynamically stop a process.

HTTP Message:

```
HTTP/1.1 POST /processes/<name>/_stop
```

The response return {"ok": true} with an http status 200 if everything is ok.

Properties:

- **name**: name of the process

Command line:

```
gafferctl stop name
```

Options

- <name>: name of the process to start

Decrement the number of OS processes This command dynamically decrease the number of OS processes for this process description to monitor in gafferd.

HTTP Message:

```
HTTP/1.1 POST /processes/<name>/_sub/<inc>
```

The response return {"ok": true} with an http status 200 if everything is ok.

Properties:

- **name**: name of the process
- **inc**: The number of new OS processes to stop

Command line:

```
gafferctl sub name inc
```

Options

- <name>: name of the process to create
- <inc>: The number of new OS processes to stop

Update a process description This command dynamically update a process monitored in gafferd. It will stop all processes withe old description before restarting them with the new settings.

HTTP Message:

```
HTTP/1.1 POST /processes/name
Content-Type: application/json

{
    "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
}
```

The response return {"ok": true} with an http status 200 if everything is ok.

It return a 409 error in case of a conflict (a process with this name has already been created.

Properties:

- **name**: name of the process
- **cmd**: program command, string)
- **args**: the arguments for the command to run. Can be a list or a string. If **args** is a string, it's splitted 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

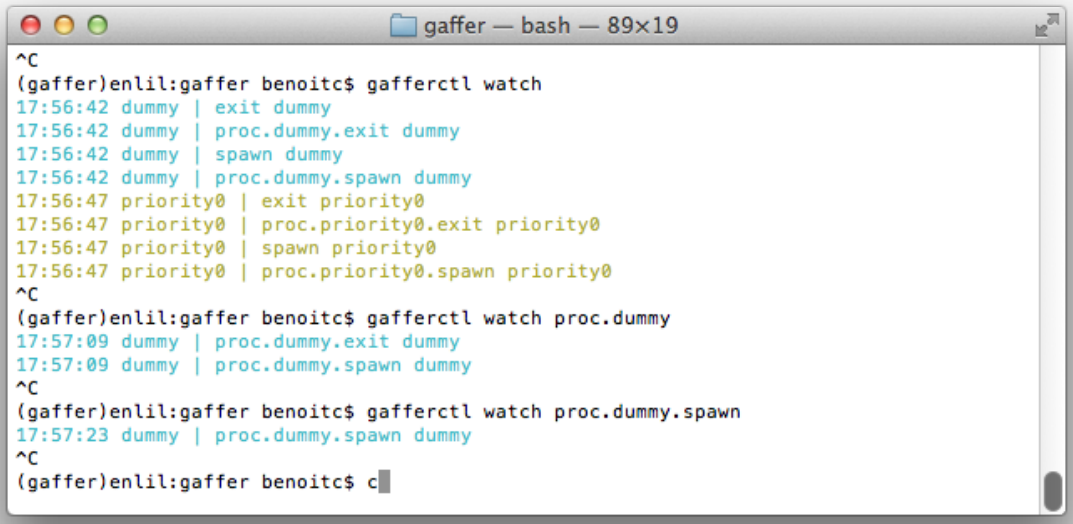
Command line:

```
gafferctl update_process [--start] name
```

Options

- <name>: name of the process to create
- <cmd>: full command line to execute in a process

Watch changes in gaffer This command allows you to watch changes n a locla or remote gaffer node.



HTTP Message

```
HTTP/1.1 GET /watch/<p1>[/<p2>/<p3>]
```

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 "http://127.0.0.1:5000/watch?feed=eventsource&heartbeat=true"
event: exit
data: {"os_pid": 3492, "exit_status": 0, "pid": 1, "event": "exit", "term_signal": 0, "name": "prior
event: exit
```

```
event: proc.priority0.exit
...
```

The path passed can be any accepted patterns by the manager :

- `create` will become `http://127.0.0.1:5000/watch/create`
- `proc.dummy` will become `http://127.0.0.1:5000/watch/proc/dummy`

...

Accepted genetic patterns

Patterns	Description
create	to follow all templates creattion
start	start all processes in a tpl
stop	all processes in a tpl are stopped
restart	restart all processes in a tpl
update	update a tpl (can happen on add/sub)
delete	a template has been removed
spawn	a new process is spawned
reap	a process is reaped
exit	a process exited
stop_pid	a process has been stopped
proc.<name>.start	process template with <name> start
proc.<name>.stop	process template with <name> stop
proc.<name>.stop_pid	a process from <name> is stopped
proc.<name>.spawn	a process from <name> is spawned
proc.<name>.exit	a process from <name> exited
proc.<name>.reap	a process from <name> has been reaped

Command line:

```
gafferctl watch <p1>[.<p2>.<p3>]
```

Note: <p1[2,3]> are the parts of the parttern separrated with a '.' .

Options:

- **heartbeat**: by default true, can be an int
- **colorize**: by default true: colorize the output

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*:

Increment the number of OS processes

This command dynamically increase the number of OS processes for this process description to monitor in gafferd.

HTTP Message:

```
HTTP/1.1 POST /processes/<name>/_add/<inc>
```

The response return {"ok": true} with an http status 200 if everything is ok.

Properties:

- **name**: name of the process
- **inc**: The number of new OS processes to start

Command line:

```
gafferctl add name inc
```

Options

- <name>: name of the process to create
- <inc>: The number of new OS processes to start

Add a process to monitor

This command dynamically add a process to monitor in gafferd.

HTTP Message:

```
HTTP/1.1 POST /processes
Content-Type: application/json
Accept: application/json

{
    "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
}
```

The response return {"ok": true} with an http status 200 if everything is ok.

It return a 409 error in case of a conflict (a process with this name has already been created.

Properties:

- **name**: name of the process
- **cmd**: program command, string)
- **args**: the arguments for the command to run. Can be a list or a string. If **args** is a string, it's splitted 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

Command line:

```
gafferctl add_process [--start] name cmd
```

Options

- <name>: name of the process to create
- <cmd>: full command line to execute in a process
- –start: start the watcher immediately

Get a process description

This command stop a process and remove it from the monitored process.

HTTP Message:

```
HTTP/1.1 DELETE /processes/<name>
```

The response return {"ok": true} with an http status 200 if everything is ok.

Properties:

- **name**: name of the process

Command line:

```
gafferctl del_process name
```

Options

- <name>: name of the process to remove

Fetch a process template

This command stop a process and remove it from the monitored process.

HTTP Message:

```
HTTP/1.1 GET /processes/<name>
```

The response return:

```
{
    "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
}
```

with an http status 200 if everything is ok.

Properties:

- **name**: name of the process
- **cmd**: program command, string)
- **args**: the arguments for the command to run. Can be a list or a string. If **args** is a string, it's splitted 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

Command line:

```
gafferctl get_process name
```

Options

- <name>: name of the process details to fetch

Send a signal to a process

This command send any signal to a process by name or id.

HTTP Message

```
HTTP/1.1 POST /processes/<name_or_id>/_signal/<signum>
```

The response *{"ok": True}* if everything was ok.

Command line:

```
gafferctl kill <name_or_id> <signum>
```

Options

- <name_or_id>: name or id of the process
- <signum>: number or name, POSIX signal number (man signal or kill for more information):

```
No     Name         Default Action       Description
1      SIGHUP       terminate process    terminal line hangup
2      SIGINT       terminate process    interrupt program
3      SIGQUIT      create core image    quit program
4      SIGILL       create core image    illegal instruction
5      SIGTRAP      create core image    trace trap
6      SIGABRT      create core image    abort program (formerly SIGIOT)
7      SIGEMT       create core image    emulate instruction executed
8      SIGFPE       create core image    floating-point exception
9      SIGKILL      terminate process    kill program
10     SIGBUS       create core image    bus error
11     SIGSEGV      create core image    segmentation violation
12     SIGSYS       create core image    non-existent system call invoked
13     SIGPIPE      terminate process    write on a pipe with no reader
14     SIGALRM      terminate process    real-time timer expired
15     SIGTERM      terminate process    software termination signal
16     SIGURG       discard signal       urgent condition present on socket
17     SIGSTOP      stop process         stop (cannot be caught or ignored)
18     SIGTSTP      stop process         stop signal generated from keyboard
19     SIGCONT      discard signal       continue after stop
20     SIGCHLD      discard signal       child status has changed
21     SIGTTIN      stop process         background read attempted from control terminal
22     SIGTTOU      stop process         background write attempted to control terminal
23     SIGIO        discard signal       I/O is possible on a descriptor (see fcntl(2))
24     SIGXCPU      terminate process    cpu time limit exceeded (see setrlimit(2))
25     SIGXFSZ      terminate process    file size limit exceeded (see setrlimit(2))
26     SIGVTALRM    terminate process    virtual time alarm (see setitimer(2))
27     SIGPROF      terminate process    profiling timer alarm (see setitimer(2))
```

```
28    SIGWINCH     discard signal       Window size change
29    SIGINFO      discard signal       status request from keyboard
30    SIGUSR1      terminate process    User defined signal 1
31    SIGUSR2      terminate process    User defined signal 2
```

Load a process from a file

Like the command `add`, his command dynamically add a process to monitor in gafferd. Informations are gathered from a file or stdin if the name of file is `-`. The file sent is a json file that have the same format described for the HTTP message.

HTTP Message:

```
HTTP/1.1 POST /processes
Content-Type: application/json
Accept: application/json

{
    "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
}
```

The response return {"ok": true} with an http status 200 if everything is ok.

It return a 409 error in case of a conflict (a process with this name has already been created.

Properties:

- **name**: name of the process
- **cmd**: program command, string)
- **args**: the arguments for the command to run. Can be a list or a string. If **args** is a string, it's splitted 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

Command line:

```
gafferctl load_process [--start] <file>
```

Options

- <name>: name of the process to create
- <file>: path to a json file or stdin –
- –start: start the watcher immediately

Example of usage:

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

Number of processes that should be launched

This command return the number of processes that should be launched

HTTP Message:

```
HTTP/1.1 GET /status/<name>
```

The response return:

```
{
    "active": true,
    "running": 1,
    "numprocesses": 1
}
```

with an http status 200 if everything is ok.

Properties:

- **name**: name of the process

Command line:

```
gafferctl numprocesses name
```

Options

- <name>: name of the process to start

Get launched process ids for a process template

This command return the list of launched process ids for a process template. Process ids are internals ids (for some reason we don't expose the system process ids)

HTTP Message:

```
HTTP/1.1 GET /processes/<name>/_pids
```

The response return:

```
{
    "ok": true,
    "pids": [1],
}
```

with an http status 200 if everything is ok.

Properties:

- **name**: name of the process

Command line:

```
gafferctl pids name
```

Options

- <name>: name of the process to start

Add a process to monitor

This command dynamically add a process to monitor in gafferd.

HTTP Message:

```
HTTP/1.1 GET /processes?running=true
```

The response return a list of processes. If running=true it will return the list of running processes by pids (pids are internal process ids).

Command line:

```
gafferctl processes [--running]
```

Options

- <name>: name of the process to create
- –running: return the list of process by pid

Number of running processes for this process description

This command return the number of processes that are currently running.

HTTP Message:

```
HTTP/1.1 GET /status/<name>
```

The response return:

```
{
    "active": true,
    "running": 1,
    "numprocesses": 1
}
```

with an http status 200 if everything is ok.

Properties:

- **name**: name of the process

Command line:

```
gafferctl running name
```

Options

- <name>: name of the process to start

Start a process

This command dynamically start a process.

HTTP Message:

```
HTTP/1.1 POST /processes/<name>/_start
```

The response return {"ok": true} with an http status 200 if everything is ok.

Properties:

- **name**: name of the process

Command line:

```
gafferctl start name
```

Options

- <name>: name of the process to start

Return the status of a process

This command dynamically add a process to monitor in gafferd.

HTTP Message:

```
HTTP/1.1 GET /status/name
Content-Type: application/json
```

The response return:

```
{
    "active": true,
    "running": 1,
    "numprocesses": 1
}
```

with an http status 200 if everything is ok.

Properties:

- **name**: name of the process

Command line:

```
gafferctl status name
```

Options

- <name>: name of the process to create

Stop a process

This command dynamically stop a process.

HTTP Message:

```
HTTP/1.1 POST /processes/<name>/_stop
```

The response return {"ok": true} with an http status 200 if everything is ok.

Properties:

- **name**: name of the process

Command line:

```
gafferctl stop name
```

Options

- <name>: name of the process to start

Decrement the number of OS processes

This command dynamically decrease the number of OS processes for this process description to monitor in gafferd.

HTTP Message:

```
HTTP/1.1 POST /processes/<name>/_sub/<inc>
```

The response return {"ok": true} with an http status 200 if everything is ok.

Properties:

- **name**: name of the process
- **inc**: The number of new OS processes to stop

Command line:

```
gafferctl sub name inc
```

Options

- <name>: name of the process to create
- <inc>: The number of new OS processes to stop

Update a process description

This command dynamically update a process monitored in gafferd. It will stop all processes withe old description before restarting them with the new settings.

HTTP Message:

```
HTTP/1.1 POST /processes/name
Content-Type: application/json

{
    "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
}
```

The response return {"ok": true} with an http status 200 if everything is ok.

It return a 409 error in case of a conflict (a process with this name has already been created.

Properties:

- **name**: name of the process
- **cmd**: program command, string)
- **args**: the arguments for the command to run. Can be a list or a string. If **args** is a string, it's splitted 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

Command line:

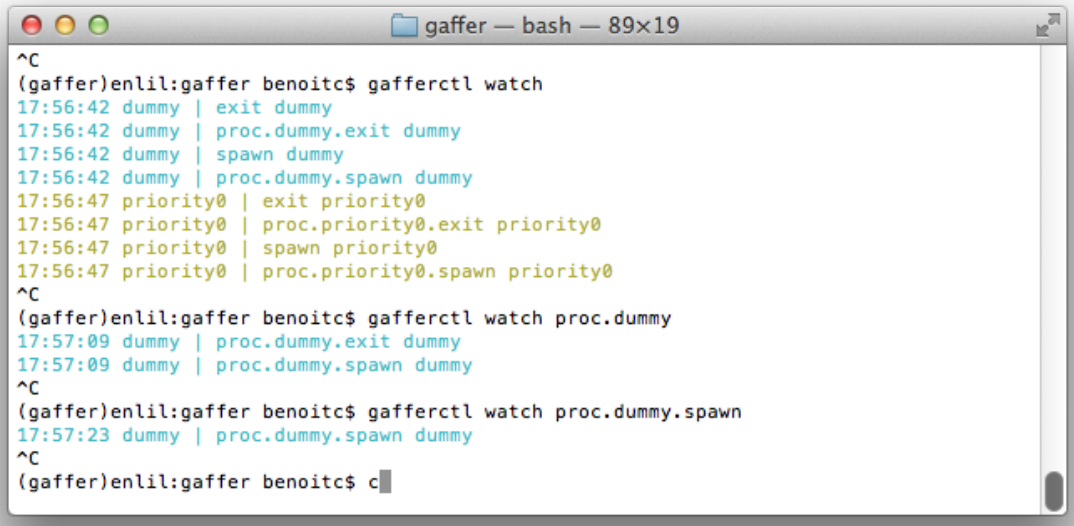
```
gafferctl update_process [--start] name
```

Options

- <name>: name of the process to create
- <cmd>: full command line to execute in a process

Watch changes in gaffer

This command allows you to watch changes n a locla or remote gaffer node.



HTTP Message

```
HTTP/1.1 GET /watch/<p1>[/<p2>/<p3>]
```

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 "http://127.0.0.1:5000/watch?feed=eventsource&heartbeat=true"
event: exit
data: {"os_pid": 3492, "exit_status": 0, "pid": 1, "event": "exit", "term_signal": 0, "name": "prior
event: exit
event: proc.priority0.exit
...
```

The path passed can be any accepted patterns by the manager :

- `create` will become `http://127.0.0.1:5000/watch/create`
- `proc.dummy` will become `http://127.0.0.1:5000/watch/proc/dummy`

...

Accepted genetic patterns

Patterns	Description
create	to follow all templates creattion
start	start all processes in a tpl
stop	all processes in a tpl are stopped
restart	restart all processes in a tpl
update	update a tpl (can happen on add/sub)
delete	a template has been removed
spawn	a new process is spawned
reap	a process is reaped
exit	a process exited
stop_pid	a process has been stopped
proc.<name>.start	process template with <name> start
proc.<name>.stop	process template with <name> stop
proc.<name>.stop_pid	a process from <name> is stopped
proc.<name>.spawn	a process from <name> is spawned
proc.<name>.exit	a process from <name> exited
proc.<name>.reap	a process from <name> has been reaped

Command line:

```
gafferctl watch <p1>[.<p2>.<p3>]
```

Note: <p1[2,3]> are the parts of the parttern separrated with a '.' .

Options:

- **heartbeat**: by default true, can be an int
- **colorize**: by default true: colorize the output

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

Webhooks allow to register an url to a specific event (or alls) and the event will be posted on this URL. Each events can triger a post on a given url.

for example to listen all create events on http://echohttp.com/echo you can add this line in the webhooks sections of the gaffer setting file:

```
[webhooks]
create = http://echohttp.com/echo you
```

Or programatically:

```
from gaffer.manager import Manager
from gaffer.webhooks import WebHooks
hooks = [("create", "http://echohttp.com/echo you ")
webhooks = WebHooks(hooks=hooks)

manager = Manager()
manager.start(apps=[webhooks])
```

This gaffer application is started like other applications in the manager. All *Gaffer events* are supported.

The webhooks Module

class gaffer.webhooks.**WebHooks**(*hooks*=[])
Bases: object

webhook app

active

close()

decref()

incref()

jobcount

maybe_start_monitor()

maybe_stop_monitor()

refcount

register_hook(*event*, *url*)
associate an url to an event

restart()

start(*loop*, *manager*)
start the webhook app

stop()
stop the webhook app, stop monitoring to events

unregister_hook(*event*, *url*)
unregister an url for this event

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.

add_process(*name*, *cmd*, ***kwargs*)

add a process to the manager. all process should be added using this function

- **name**: name of the process
- **cmd**: program command, string)
- **args**: the arguments for the command to run. Can be a list or a string. If **args** is a string, it's splitted 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 stdout & 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.

- **graceful_timeout**: graceful time before we send a SIGKILL to the process (which definitely kill it). By default 30s. This is a time we let to a process to exit cleanly.

get_group(*groupname*)
return list of named process of this group

get_groups()
return the groups list

get_process(*name_or_pid*)

get_process_id()
generate a process id

get_process_info(*name*)
get process info

get_process_state(*name*)

get_process_stats(*name_or_id*)
return process stats for a process template or a process id

get_process_status(*name*)
return the process status:

```
{
  "active":  str,
  "running": int,
  "max_processes": int
}
```

- **active** can be *active* or *stopped*
- **running**: the number of actually running OS processes using this template.
- **max_processes**: The maximum number of processes that should run. It is is normally the same than the **runnin** value.

manage_process(*name*)

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

on(*evtype*, *listener*)
subscribe to the manager event *eventype*

'on' is an alias to this function

once(*evtype*, *listener*)
subscribe to the manager event *eventype*

'on' is an alias to this function

processes_stats()
iterator returning all processes stats

remove_group(*groupname*)
remove a group and all its processes. All processes are stopped

remove_process(*name*)
remove the process and its config from the manager

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

restart_group(*groupname*)
restart all processes in a group

restart_process(*name*)
restart a process

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.

running_processes()
return running processes

send_signal(*name_or_id*, *signum*)
send a signal to a process or all processes contained in a state

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

start_group(*groupname*)
start all process templates of the group

start_process(*name*)

start_processes()
start all processes

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

stop_group(*groupname*)
stop all processes templates of the group

stop_process(*name_or_id*)
stop a process by name or id

If a name is given all processes associated to this name will be removed and the process is marked at stopped. If the internal process id is givien, only the process with this id will be stopped

stop_processes()
stop all processes in the manager

subscribe(*evtype*, *listener*)
subscribe to the manager event *eventype*

'on' is an alias to this function

subscribe_once(*evtype*, *listener*)
subscribe once to the manager event *eventype*

'once' is an alias to this function

ttin(*name*, *i=1*)
increase the number of system processes for a state. Change is handled once the event loop is idling

ttou(*name*, *i=1*)
decrease the number of system processes for a state. Change is handled once the event loop is idling

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

unsubscribe(*evtype*, *listener*)
unsubscribe from the event *eventype*

update_process(*name*, *cmd*, ***kwargs*)
update a process information.

When a process is updated, all current processes are stopped then the state is updated and new processes with new info are started

wakeup()

process Module

The process module wrap a process and IO redirection

class gaffer.process.**Process**(*loop*, *id*, *name*, *cmd*, *group=None*, *args=None*, *env=None*, *uid=None*, *gid=None*, *cwd=None*, *detach=False*, *shell=False*, *redirect_output=[]*, *redirect_input=False*, *custom_streams=[]*, *custom_channels=[]*, *on_exit_cb=None*)

Bases: object

class wrapping a process

Args:

- **loop**: main application loop (a pyuv Loop instance)
- **name**: name of the process
- **cmd**: program command, string)
- **args**: the arguments for the command to run. Can be a list or a string. If **args** is a string, it's splitted 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)
- **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.
- **custom_streams**: list of additional streams that should be created and passed to process. This is a list of streams labels. They become available through streams attribute.
- **custom_channels**: list of additional channels that should be passed to process.

active

close()

closed

info
return the process info. If the process is monitored it return the last informations stored asynchronously by the watcher

kill(*signum*)
send a signal to the process

monitor(*listener=None*)
start to monitor the process

Listener can be any callable and receive *("stat", process_info)*

monitor_io(*io_label*, *listener*)
subscribe to registered IO events

pid
return the process pid

spawn()
spawn the process

status
return the process status

stop()
stop the process

unmonitor(*listener*)
stop monitoring this process.

listener is the callback passed to the monitor function previously.

unmonitor_io(*io_label*, *listener*)
unsubscribe to the IO event

write(*data*)
send data to the process via stdin

writelines(*data*)
send data to the process via stdin

class gaffer.process.**ProcessWatcher**(*loop*, *pid*)
Bases: object

object to retrieve process stats

active

refresh(*interval=0*)

stop(*all_events=False*)

subscribe(*listener*)

subscribe_once(*listener*)

unsubscribe(*listener*)

class gaffer.process.**RedirectIO**(*loop*, *process*, *stdio=[]*)
Bases: object

pipes_count = 2

start()

stdio

stop(*all_events=False*)

subscribe(*label*, *listener*)

unsubscribe(*label*, *listener*)

class gaffer.process.**RedirectStdin**(*loop*, *process*)
Bases: object

redirect stdin allows multiple sender to write to same pipe

start()

stop(*all_events=False*)

write(*data*)

writelines(*data*)

class gaffer.process.**Stream**(*loop*, *process*, *id*)
Bases: gaffer.process.RedirectStdin

create custom stdio

start()

subscribe(*listener*)

unsubscribe(*listener*)

gaffer.process.**get_process_info**(*process=None*, *interval=0*)
Return information about a process. (can be an pid or a Process object)

If process is None, will return the information about the current process.

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=200*)
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

Webhooks allow to register an url to a specific event (or alls) and the event will be posted on this URL. Each events can triger a post on a given url.

for example to listen all create events on http://echohttp.com/echo you can add this line in the webhooks sections of the gaffer setting file:

```
[webhooks]
create = http://echohttp.com/echo you
```

Or programatically:

```
from gaffer.manager import Manager
from gaffer.webhooks import WebHooks
hooks = [("create", "http://echohttp.com/echo you ")
webhooks = WebHooks(hooks=hooks)

manager = Manager()
manager.start(apps=[webhooks])
```

This gaffer application is started like other applications in the manager. All *Gaffer events* are supported.

The `webhooks` Module

class `gaffer.webhooks.`**`WebHooks`**(*hooks*=[ ])
Bases: `object`

webhook app

`active`

`close`()

`decref`()

`incref`()

`jobcount`

`maybe_start_monitor`()

`maybe_stop_monitor`()

`refcount`

`register_hook`(*event*, *url*)
associate an url to an event

`restart`()

`start`(*loop*, *manager*)
start the webhook app

`stop`()
stop the webhook app, stop monitoring to events

`unregister_hook`(*event*, *url*)
unregister an url for this event

`procfile` Module

module to parse and manage a Procfile

class `gaffer.procfile.`**`Procfile`**(*procfile*, *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_env(*envs=[]*)
build the procfile environment from a list of procfiles

get_groupname()

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

parse_cmd(*v*)

processes()
iterator over the configuration

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.**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.**get_maxfd**()

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

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

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

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

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(*all_fds=False*, *all_handlers=False*)

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(*run_loop=True*)

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

Gaffer provides you a simple Client to control a gaffer node via HTTP.

Example of usage:

```
import pyuv

from gaffer.httpclient import Server

# initialize a loop
loop = pyuv.Loop.default_loop()

s = Server("http://localhost:5000", loop=loop)

# add a process without starting it
process = s.add_process("dummy", "/some/path/to/dummy/script", start=False)

# start a process
process.start()

# increase the number of process by 2 (so 3 will run)
process.add(2)

# stop all processes
process.stop()

loop.run()
```

class gaffer.httpclient.**EventsourceClient**(*loop*, *url*, ***kwargs*)
Bases: object

simple client to fetch Gaffer streams using the eventsource stream.

Example of usage:

```
loop = pyuv.Loop.default_loop()

def cb(event, data):
    print(data)

# create a client
url = http://localhost:5000/streams/1/stderr?feed=continuous'
client = EventSourceClient(loop, url)

# subscribe to the stderr event
client.subscribe("stderr", cb)
```

```
# start the client
client.start()
```

render(*event*, *data*)

run()

start()

stop()

subscribe(*event*, *listener*)

subscribe_once(*event*, *listener*)

unsubscribe(*event*, *listener*)

exception gaffer.httpclient.**GafferConflict**
Bases: exceptions.Exception

exption raised on HTTP 409

exception gaffer.httpclient.**GafferNotFound**
Bases: exceptions.Exception

exception raised on HTTP 404

class gaffer.httpclient.**HTTPClient**(*async_client_class=None*, *loop=None*, ***kwargs*)
Bases: object

A blocking HTTP client.

This interface is provided for convenience and testing; most applications that are running an IOLoop will want to use *AsyncHTTPClient* instead. Typical usage looks like this:

```
http_client = httpclient.HTTPClient()
try:
    response = http_client.fetch("http://www.friendpaste.com/")
    print response.body
except httpclient.HTTPError as e:
    print("Error: %s" % e)
```

close()
Closes the HTTPClient, freeing any resources used.

fetch(*request*, ***kwargs*)
Executes a request, returning an *HTTPResponse*.

The request may be either a string URL or an *HTTPRequest* object. If it is a string, we construct an *HTTPRequest* using any additional kwargs: `HTTPRequest(request, **kwargs)`

If an error occurs during the fetch, we raise an *HTTPError*.

class gaffer.httpclient.**Process**(*server*, *process*)
Bases: object

Process object. Represent a remote process state

active
return True if the process is active

add(*num=1*)
increase the number of processes for this template

info()
return the process info dict

numprocesses
return the maximum number of processes that can be launched for this template

pids
return a list of running pids

restart()
restart the process

running
return the number of processes running for this template

signal(*num_or_str*)
send a signal to all processes of this template

start()
start the process if not started, spawn new processes

stats()

status()
Return the status

```
{
    "active": true,
    "running": 1,
    "numprocesses": 1
}
```

stop()
stop the process

sub(*num=1*)
decrease the number of processes for this template

class gaffer.httpclient.**ProcessId**(*server*, *pid*, *process*)
Bases: object

Process Id object. It represent a pid

active
return True if the process is active

signal(*num_or_str*)
Send a signal to the pid

stop()
stop the process

class gaffer.httpclient.**Server**(*uri*, *loop=None*, ***options*)
Bases: object

Server, main object to connect to a gaffer node. Most of the calls are blocking. (but running in the loop)

add_process(*name*, *cmd*, ***kwargs*)
add a process. Use the same arguments as in save_process.

If a process with the same name is already registred a *GafferConflict* exception is raised.

get_group(*name*)
return the list of all process templates of this group

get_process(*name_or_id*)
get a process by name or id.

If id is given a ProcessId instance is returned in other cases a Process instance is returned.

get_watcher(*heartbeat='true'*)
return a watcher to listen on /watch

groups()
return the list of all groups

is_process(*name*)
is the process exists ?

json_body(*resp*)

processes()
get list of registered processes

remove_group(*name*)
remove the group and all process templates of the group

remove_process(*name*)
Stop a process and remove it from the managed processes

request(*method*, *path*, *headers=None*, *body=None*, ***params*)

restart_group(*name*)
restart all process templates of the group

running()
get list of running processes by pid

save_process(*name*, *cmd*, ***kwargs*)
save a process.

Args:

- **name**: name of the process
- **cmd**: program command, string)
- **args**: the arguments for the command to run. Can be a list or a string. If **args** is a string, it's splitted 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

If *_force_update=True* is passed, the existing process template will be overwritten.

send_signal(*name_or_id*, *num_or_str*)
Send a signal to the pid or the process name

start_group(*name*)
start all process templates of the group

stop_group(*name*)
stop all process templates of the group

update_process(*name*, *cmd*, ***kwargs*)
update a process.

version
get gaffer version

class gaffer.httpclient.**Watcher**(*loop*, *url*, ***kwargs*)
Bases: gaffer.httpclient.EventsourceClient

simple EventsourceClient wrapper that decode the JSON to a python object

render(*event*, *data*)

gaffer.httpclient.**encode**(*v*, *charset='utf8'*)

gaffer.httpclient.**make_uri**(*base*, **args*, ***kwargs*)
Assemble a uri based on a base, any number of path segments, and query string parameters.

gaffer.httpclient.**url_encode**(*obj*, *charset='utf8'*, *encode_keys=False*)

gaffer.httpclient.**url_quote**(*s*, *charset='utf-8'*, *safe='/:'*)
URL encode a single string with a given encoding.

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

class gaffer.http_handler.**HttpEndpoint**(*uri='127.0.0.1:5000'*, *backlog=128*, *ssl_options=None*)
Bases: object

restart()

start(*loop*, *app*)

stop()

class gaffer.http_handler.**HttpHandler**(*endpoints=[]*, *handlers=None*, ***settings*)
Bases: object

simple gaffer application that gives an HTTP API access to gaffer.

This application can listen on multiple endpoints (tcp or unix sockets) with different options. Each endpoint can also listen on different interfaces

restart()

start(*loop*, *manager*)

stop()

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 = ['spawn', 'reap', 'exit', 'stop_pid']

restart(*start*)

start(*loop*, *manager*)

stop()

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*, **args*)

handle_reload(*handle*, **args*)

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

TWO

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

g