

---

# **Circus Documentation**

***Release 0.5***

**Mozilla Foundation**

July 06, 2012



# CONTENTS





Circus is a process & socket manager. It can be used to monitor and control processes and sockets.

Circus can be driven via a command-line interface or programmatically through its python API.

It shares some of the goals of [Supervisord](#), [BluePill](#) and [Daemontools](#). If you are curious about what Circus brings compared to other projects, read [Why should I use Circus instead of X ?](#).

Circus is designed using [ZeroMQ](http://www.zeromq.org/) [<http://www.zeromq.org/>](http://www.zeromq.org/). See [Design](#) for more details.

---

**Note:** Circus doesn't secure its messages when sending information through ZeroMQ. Before running Circus, make sure you read the [Security](#) page.

---

To install it, check out [Installing Circus](#)



# RUNNING CIRCUS

Circus provides a command-line script call **circusd** that can be used to manage one or more *watchers*. Each watcher can have one or more running *processes*.

Circus' command-line tool is configurable using an ini-style configuration file. Here is a minimal example:

```
[circus]
check_delay = 5
endpoint = tcp://127.0.0.1:5555

[watcher:myprogram]
cmd = python
args = -u myprogram.py $WID
warmup_delay = 0
numprocesses = 5

[watcher:anotherprogram]
cmd = another_program
numprocesses = 2
```

The file is then run using *circusd*:

```
$ circusd example.ini
```

Besides processes, Circus can also bind sockets. Since every process managed by Circus is a child of the main Circus daemon, that means any program that's controlled by Circus can use those sockets.

Running a socket is as simple as adding a *socket* section in the config file:

```
[socket:mysocket]
host = localhost
port = 8080
```

To learn more about sockets, see *Circus Sockets*.

To understand why it's a killer feature, read *Circus stack v.s. Classical stack*.





# CONTROLLING CIRCUS

Circus provides two command-line tools to manage your running daemon:

- *circusctl*, a management console you can use it to perform actions such as adding or removing *workers*
- *circus-top*, a top-like console you can use to display the memory and cpu usage of your running Circus.

To learn more about these, see *Command-line tools*

Circus also offers a small web application that can connect to a running Circus daemon and let you monitor and interact with it.

Running the web application is as simple as adding an **httpd** option in the ini file:

```
[circus]
httpd = True
```

Or if you want, you can run it as a standalone process with:

```
$ circushttpd
```

By default, **circushttpd** runs on the 8080 port.

To learn more about this feature, see *The Web Console*



# DEVELOPING WITH CIRCUS

Circus provides high-level classes and functions that will let you manage processes in your own applications.

For example, if you want to run four processes forever, you could write:

```
from circus import get_arbiter

arbiter = get_arbiter("myprogram", 4)
try:
    arbiter.start()
finally:
    arbiter.stop()
```

This snippet will run four instances of *myprogram* and watch them for you, restarting them if they die unexpectedly.

To learn more about this, see [Circus Library](#)



# EXTENDING CIRCUS

It's easy to extend Circus to create a more complex system, by listening to all the **circusd** events via its pub/sub channel, and driving it via commands.

That's how the flapping feature works for instance: it listens to all the processes dying, measures how often it happens, and stops the incriminated watchers after too many restarts attempts.

Circus comes with a plugin system to help you write such extensions, and a few built-in plugins you can reuse.

See *[The Plugin System](#)*.



# MORE DOCUMENTATION

## 5.1 Installing Circus

Use pip:

```
$ pip install circus
```

Or download the archive on PyPI, extract and install it manually with:

```
$ python setup.py install
```

If you want to try out Circus, see the *Examples*.

## 5.2 More on Requirements

Circus uses:

- Python 2.6, 2.7 (3.x needs to be tested)
- zeromq >= 2.10

And on Python side:

- pyzmq 2.2.0
- iowait 0.1
- psutil 0.4.1

You can install all the py dependencies with the pip-requirements.txt file we provide manually, or just install Circus and have the latest versions of those libraries pulled for you:

```
$ pip install -r pip-requirements.txt
```

If you want to run the Web console you will need more things:

- Mako 0.7.0
- MarkupSafe 0.15
- bottle 0.10.9
- anyjson 0.3.1
- gevent 0.13.7
- gevent-socketio 0.3.5-beta

- gevent-websocket 0.3.6
- greenlet 0.3.4
- beaker 1.6.3
- <http://github.com/tarekziade/gevent-zeromq>

Those can be installed with:

```
$ pip install -r web-requirements.txt
```

## 5.3 Configuration

Circus can be configured using an ini-style configuration file.

Example:

```
[circus]
check_delay = 5
endpoint = tcp://127.0.0.1:5555
pubsub_endpoint = tcp://127.0.0.1:5556
include = /path/to/configs/*.enabled.ini

[watcher:myprogram]
cmd = python
args = -u myprogram.py $(WID) $(ENV.VAR)
warmup_delay = 0
numprocesses = 5

# will push in test.log the stream every 300 ms
stdout_stream.class = FileStream
stdout_stream.filename = test.log
stdout_stream.refresh_time = 0.3

[plugin:statsd]
use = circus.plugins._statsd.StatsdEmitter
host = localhost
port = 8125
sample_rate = 1.0
application_name = example

[socket:web]
host = localhost
port = 8080
```

### 5.3.1 circus - single section

**endpoint** The ZMQ socket used to manage Circus via **circusctl**. (default: *tcp://127.0.0.1:5555*)

**pubsub\_endpoint** The ZMQ PUB/SUB socket receiving publications of events. (default: *tcp://127.0.0.1:5556*)

**stats\_endpoint** The ZMQ PUB/SUB socket receiving publications of stats. If not configured, this feature is deactivated. (default: *tcp://127.0.0.1:5557*)

**check\_delay** The polling interval in seconds for the ZMQ socket. (default: 5)



**include** List of config files to include. (default: None). You can use wildcards (\*) to include particular schemes for your files.

**include\_dir** List of config directories. All files matching \*.ini under each directory will be included. (default: None)

**stream\_backend** Defines the type of backend to use for the streaming. Possible values are **thread** or **gevent**. (default: thread)

**warmup\_delay** The interval in seconds between two watchers start. (default: 0)

**httpd** If set to True, Circus runs the circushttpd daemon. (default: False)

**httpd\_host** The host ran by the circushttpd daemon. (default: localhost)

**httpd\_port** The port ran by the circushttpd daemon. (default: 8080)

---

**Note:** If you use the gevent backend for **stream\_backend**, you need to install the forked version of gevent\_zmq that's located at <https://github.com/tarekziade/gevent-zeromq> because it contains a fix that has not made it upstream yet.

---

### 5.3.2 watcher:NAME - as many sections as you want

**NAME** The name of the watcher. This name is used in **circusctl**

**cmd** The executable program to run.

**args** Command-line arguments to pass to the program. You can use the python format syntax here to build the parameters. Environment variables are available, as well as the worker id and the environment variables that you passed, if any, with the “env” parameter. See *Formating the commands and arguments with dynamic variables* for more information on this.

**shell** If True, the processes are run in the shell (default: False)

**working\_dir** The working dir for the processes (default: None)

**uid** The user id or name the command should run with. (The current uid is the default).

**gid** The group id or name the command should run with. (The current gid is the default).

**env** The environment passed to the processes (default: None)

**copy\_env** If set to true, the local environment variables will be copied and passed to the workers when spawning them. (Default: False)

**warmup\_delay** The delay (in seconds) between running processes.

**numprocesses** The number of processes to run for this watcher.

**rlimit\_LIMIT** Set resource limit LIMIT for the watched processes. The config name should match the RLIMIT\_\* constants (not case sensitive) listed in the [Python resource module reference](#). For example, the config line ‘rlimit\_nofile = 500’ sets the maximum number of open files to 500.

**stderr\_stream.class** A fully qualified Python class name that will be instantiated, and will receive the **stderr** stream of all processes in its `__call__()` method.

Circus provides two classes you can use without prefix:

- `FileStream`: writes in a file
- `QueueStream`: write in a memory Queue
- `StdoutStream`: writes in the stdout

**stderr\_stream.\*** All options starting with *stderr\_stream*. other than *class* will be passed the constructor when creating an instance of the class defined in **stderr\_stream.class**.

**stdout\_stream.class** A fully qualified Python class name that will be instantiated, and will receive the **stdout** stream of all processes in its `__call__()` method. Circus provides two classes you can use without prefix:

- `FileStream`: writes in a file
- `QueueStream`: write in a memory Queue
- `StdoutStream`: writes in the stdout

**stdout\_stream.\*** All options starting with *stdout\_stream*. other than *class* will be passed the constructor when creating an instance of the class defined in **stdout\_stream.class**.

**send\_hup** if True, a process reload will be done by sending the SIGHUP signal. Defaults to False.

**max\_retry** The number of times we attempt to start a process, before we abandon and stop the whole watcher. Defaults to 5.

**priority** Integer that defines a priority for the watcher. When the Arbiter do some operations on all watchers, it will sort them with this field, from the bigger number to the smallest. Defaults to 0.

**singleton** If set to True, this watcher will have at the most one process. Defaults to False.

**use\_sockets** If set to True, this watcher will be able to access defined sockets via their file descriptors. If False, all parent fds are closed when the child process is forked. Defaults to False.

### 5.3.3 plugin:NAME - as many sections as you want

**use** The fully qualified name that points to the plugin class.

**anything else** Every other key found in the section is passed to the plugin constructor in the **config** mapping.

### 5.3.4 socket:NAME - as many sections as you want

**host** The host of the socket. Defaults to 'localhost'

**port** The port. Defaults to 8080.

**family** The socket family. Can be 'AF\_UNIX', 'AF\_INET' or 'AF\_INET6'. Defaults to 'AF\_INET'.

**type** The socket type. Can be 'SOCK\_STREAM', 'SOCK\_DGRAM', 'SOCK\_RAW', 'SOCK\_RDM' or 'SOCK\_SEQPACKET'. Defaults to 'SOCK\_STREAM'.

Once a socket is created, the `${socket:NAME}` string can be used in the command of a watcher. Circus will replace it by the FD value.

### 5.3.5 Formating the commands and arguments with dynamic variables

As you may have seen, it is possible to pass some information that are computed dynamically when running the processes. Among other things, you can get the worker id (WID) and all the options that are passed to the `Process`. Additionally, it is possible to access the options passed to the `Watcher` which instantiated the process.

For instance, if you want to access some variables that are contained in the environment, you would need to do it with a setting like this:

```
cmd = "make-me-a-coffee --sugar ${CIRCUS.ENV.SUGAR_AMOUNT} "
```

This works with both *cmd* and *args*.

**Important:**

- All variables are prefixed with *circus*.
- The replacement is case insensitive.

## 5.4 Command-line tools

### 5.4.1 circus-top

*circus-top* is a top-like console you can run to watch live your running Circus system. It will display the CPU, Memory usage and socket hits if you have some.

Example of output:

```
-----
circusd-stats
  PID          CPU (%)          MEMORY (%)
14252          0.8             0.4
              0.8 (avg)        0.4 (sum)

dummy
  PID          CPU (%)          MEMORY (%)
14257          78.6             0.1
14256          76.6             0.1
14258          74.3             0.1
14260          71.4             0.1
14259          70.7             0.1
              74.32 (avg)      0.5 (sum)
-----
```

*circus-top* is a read-only console. If you want to interact with the system, use *circusctl*.

### 5.4.2 circusctl

*circusctl* can be used to run any command listed in [Commands](#) . For example, you can get a list of all the watchers, you can do

```
$ circusctl list
```

## 5.5 Commands

At the epicenter of circus lives the command systems. *circusctl* is just a zeromq client, and if needed you can drive programmatically the Circus system by writing your own zmq client.

All messages are Json mappings.

For each command below, we provide a usage example with *circusctl* but also the input / output zmq messages.

### 5.5.1 circus-ctl commands

- **add:** *Add a watcher*
- **decr:** *Decrement the number of processes in a watcher*
- **dstats:** *Get circusd stats*
- **get:** *Get the value of a watcher option*
- **globaloptions:** *Get the arbiter options*
- **incr:** *Increment the number of processes in a watcher*
- **list:** *Get list of watchers or processes in a watcher*
- **listen:** *Suscribe to a watcher event*
- **listsockets:** *Get the list of sockets*
- **numprocesses:** *Get the number of processes*
- **numwatchers:** *Get the number of watchers*
- **options:** *Get the value of a watcher option*
- **quit:** *Quit the arbiter immediately*
- **reload:** *Reload the arbiter or a watcher*
- **restart:** *Restart the arbiter or a watcher*
- **rm:** *Remove a watcher*
- **set:** *Set a watcher option*
- **signal:** *Send a signal*
- **start:** *Start the arbiter or a watcher*
- **stats:** *Get process infos*
- **status:** *Get the status of a watcher or all watchers*
- **stop:** *Stop the arbiter or a watcher*

#### Add a watcher

This command add a watcher dynamically to a arbiter.

#### ZMQ Message

```
{
  "command": "add",
  "properties": {
    "cmd": "/path/to/commandline --option"
    "name": "nameofwatcher"
    "args": [],
    "options": {},
    "start": false
  }
}
```

A message contains 2 properties:

- `cmd`: Full command line to execute in a process
- `args`: array, arguments passed to the command (optional)
- `name`: name of watcher
- `options`: options of a watcher
- `start`: start the watcher after the creation

The response return a status “ok”.

### Command line

```
$ circusctl add [--start] <name> <cmd>
```

### Options

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

### Decrement the number of processes in a watcher

This command decrement the number of processes in a watcher by -1.

### ZMQ Message

```
{
  "command": "decr",
  "properties": {
    "name": "<watchername>",
    "nb": <nbprocess>
  }
}
```

The response return the number of processes in the ‘numprocesses’ property:

```
{ "status": "ok", "numprocesses": <n>, "time", "timestamp" }
```

### Command line

```
$ circusctl descr <name> [<nbprocess>]
```

### Options

- `<name>`: name of the watcher
- `<nbprocess>`: the number of processes to remove.

## Get circusd stats

You can get at any time some statistics about circusd with the dstat command.

### ZMQ Message

To get the circusd stats, simply run:

```
{
  "command": "dstats"
}
```

The response returns a mapping the property “infos” containing some process informations:

```
{
  "info": {
    "children": [],
    "cmdline": "python",
    "cpu": 0.1,
    "ctime": "0:00.41",
    "mem": 0.1,
    "mem_info1": "3M",
    "mem_info2": "2G",
    "nice": 0,
    "pid": 47864,
    "username": "root"
  },
  "status": "ok",
  "time": 1332265655.897085
}
```

## Command Line

```
$ circusctl dstats
```

## Get the value of a watcher option

This command return the watchers options values asked.

### ZMQ Message

```
{
  "command": "get",
  "properties": {
    "keys": ["key1", "key2"]
    "name": "nameofwatcher"
  }
}
```

A response contains 2 properties:

- keys: list, The option keys for which you want to get the values
- name: name of watcher

The response return an object with a property “options” containing the list of key/value returned by circus.

eg:

```
{
  "status": "ok",
  "options": {
    "flapping_window": 1,
    "times": 2
  },
  'time': 1332202594.754644
}
```

See [Options](#) for for a description of options enabled?

### Command line

```
$ circusctl get <name> <key> <value> <key1> <value1>
```

### Get the arbiter options

This command return the arbiter options

### ZMQ Message

```
{
  "command": "globaloptions",
  "properties": {
    "key1": "vall",
    ..
  }
}
```

A message contains 2 properties:

- **keys:** list, The option keys for which you want to get the values

The response return an object with a property “options” containing the list of key/value returned by circus.

eg:

```
{
  "status": "ok",
  "options": {
    "check_delay": 1,
    ...
  },
  'time': 1332202594.754644
}
```

### Command line

```
$ circusctl globaloptions
```

### Options

Options Keys are:

- `endpoint`: the controller ZMQ endpoint
- `pubsub_endpoint`: the pubsub endpoint
- `check_delay`: the delay between two controller points

### Increment the number of processes in a watcher

This comment increment the number of processes in a watcher by +1.

#### ZMQ Message

```
{
  "command": "incr",
  "properties": {
    "name": "<watchername>",
    "nb": <nbprocess>
  }
}
```

The response return the number of processes in the 'numprocesses' property:

```
{ "status": "ok", "numprocesses": <n>, "time", "timestamp" }
```

### Command line

```
$ circusctl incr <name> [<nbprocess>]
```

### Options

- `<name>`: name of the watcher.
- `<nbprocess>`: the number of processes to add.

### Get list of watchers or processes in a watcher

#### ZMQ Message

To get the list of all the watchers:

```
{
  "command": "list",
}
```

To get the list of active processes in a watcher:



```
{
  "command": "list",
  "properties": {
    "name": "nameofwatcher",
  }
}
```

The response return the list asked. the mapping returned can either be 'watchers' or 'pids' depending the request.

### Command line

```
$ circusctl list [<name>]
```

### Suscribe to a watcher event

#### ZMQ

At any moment you can suscribe to circus event. Circus provide a PUB/SUB feed on which any clients can suscribe. The subscriber endpoint URI is set in the circus.ini configuration file.

Events are pubsub topics:

Events are pubsub topics:

- *watcher.<watchername>.reap*: when a process is reaped
- *watcher.<watchername>.spawn*: when a process is spawned
- *watcher.<watchername>.kill*: when a process is killed
- *watcher.<watchername>.updated*: when watcher configuration is updated
- *watcher.<watchername>.stop*: when a watcher is stopped
- *watcher.<watchername>.start*: when a watcher is started

All events messages are in a json.

### Command line

The client has been updated to provide a simple way to listen on the events:

```
circusctl list [<topic>, ...]
```

#### Example of result:

```
$ circusctl listen tcp://127.0.0.1:5556
watcher.refuge.spawn: {u'process_id': 6, u'process_pid': 72976,
                       u'time': 1331681080.985104}
watcher.refuge.spawn: {u'process_id': 7, u'process_pid': 72995,
                       u'time': 1331681086.208542}
watcher.refuge.spawn: {u'process_id': 8, u'process_pid': 73014,
                       u'time': 1331681091.427005}
```

## Get the list of sockets

### ZMQ Message

To get the list of sockets:

```
{
  "command": "listsockets",
}
```

The response return a list of json mappings with keys for fd, name, host and port.

### Command line

```
$ circusctl listsockets
```

## Get the number of processes

Get the number of processes in a watcher or in a arbiter

### ZMQ Message

```
{
  "command": "numprocesses",
  "propeties": {
    "name": "<watchername>"
  }
}
```

The response return the number of processes in the ‘numprocesses’ property:

```
{ "status": "ok", "numprocesses": <n>, "time", "timestamp" }
```

If the property name isn’t specified, the sum of all processes managed is returned.

### Command line

```
$ circusctl numprocesses [<name>]
```

### Options

- <name>: name of the watcher

## Get the number of watchers

Get the number of watchers in a arbiter

### ZMQ Message

```
{
  "command": "numwatchers",
}
```

The response return the number of watchers in the ‘numwatchers’ property:

```
{ "status": "ok", "numwatchers": <n>, "time", "timestamp" }
```

### Command line

```
$ circusctl numwatchers
```

### Get the value of a watcher option

This command return the watchers options values asked.

### ZMQ Message

```
{
  "command": "options",
  "properties": {
    "name": "nameofwatcher",
    "key1": "vall1",
    ..
  }
}
```

A message contains 2 properties:

- keys: list, The option keys for which you want to get the values
- name: name of watcher

The response return an object with a property “options” containing the list of key/value returned by circus.

eg:

```
{
  "status": "ok",
  "options": {
    "flapping_window": 1,
    "flapping_attempts": 2,
    ...
  },
  'time': 1332202594.754644
}
```

### Command line

```
$ circusctl options <name>
```

### Options

- <name>: name of the watcher

Options Keys are:

- numprocesses: integer, number of processes
- warmup\_delay: integer or number, delay to wait between process spawning in seconds
- working\_dir: string, directory where the process will be executed
- uid: string or integer, user ID used to launch the process
- gid: string or integer, group ID used to launch the process
- send\_hup: boolean, if TRU the signal HUP will be used on reload
- shell: boolean, will run the command in the shell environment if true
- cmd: string, The command line used to launch the process
- env: object, define the environnement in which the process will be launch
- flapping\_attempts: integer, number of times we try to relaunch a process in the flapping\_window time before we stop the watcher during the retry\_in time.
- flapping\_window: integer or number, times in seconds in which we test the number of process restart.
- retry\_in: integer or number, time in seconds we wait before we retry to launch the process if the maximum number of attempts has been reach.
- max\_retry: integer, The maximum of retries loops
- graceful\_timeout: integer or number, time we wait before we definitely kill a process.
- priority: used to sort watchers in the arbiter
- singleton: if True, a singleton watcher.

### Quit the arbiter immediately

When the arbiter receive this command, the arbiter exit.

### ZMQ Message

```
{  
    "command": "quit"  
}
```

The response return the status “ok”.

### Command line

```
$ circusctl quit
```

### Reload the arbiter or a watcher

This command reload all the process in a watcher or all watchers. If a the option `send_hup` is set to true in a watcher then the HUP signal will be sent to the process. A graceful reload follow the following process:

1. Send a SIGQUIT signal to a process
2. Wait until graceful timeout
3. Send a SIGKILL signal to the process to make sure it is finally killed.

#### ZMQ Message

```
{
  "command": "reload",
  "propeties": {
    "name": '<name>',
    "graceful": true
  }
}
```

The response return the status “ok”. If the property `graceful` is set to true the processes will be exited gracefully.

If the property `name` is present, then the reload will be applied to the watcher.

#### Command line

```
$ circusctl reload [<name>] [--terminate]
```

#### Options

- `<name>`: name of the watcher
- `--terminate`; quit the node immediately

### Restart the arbiter or a watcher

This command restart all the process in a watcher or all watchers. This funtion simply stop a watcher then restart it.

#### ZMQ Message

```
{
  "command": "restart",
  "propeties": {
    "name": '<name>'
  }
}
```

The response return the status “ok”.

If the property `name` is present, then the reload will be applied to the watcher.

### Command line

```
$ circusctl restart [<name>] [--terminate]
```

### Options

- <name>: name of the watcher
- --terminate; quit the node immediately

### Remove a watcher

This command remove a watcher dynamically from the arbiter. The watchers are gracefully stopped.

### ZMQ Message

```
{
  "command": "rm",
  "properties": {
    "name": "nameofwatcher",
  }
}
```

A message contains 1 property:

- name: name of watcher

The response return a status “ok”.

### Command line

```
$ circusctl rm <name>
```

### Options

- <name>: name of the watcher to create

### Set a watcher option

### ZMQ Message

```
{
  "command": "set",
  "properties": {
    "name": "nameofwatcher",
    "key1": "val1",
    ..
  }
}
```

The response return the status “ok”. See the command Options for a list of key to set.

## Command line

```
$ circusctl set <name> <key1> <value1> <key2> <value2>
```

## Send a signal

This command allows you to send a signal to all processes in a watcher, a specific process in a watcher or its children.

### ZMQ Message

To send a signal to all the processes for a watcher:

```
{
  "command": "signal",
  "property": {
    "name": <name>,
    "signal": <signal>
  }
}
```

To send a signal to a process:

```
{
  "command": "signal",
  "property": {
    "name": <name>,
    "pid": <processid>,
    "signal": <signal>
  }
}
```

An optionnal property “children” can be used to send the signal to all the children rather than the process itself:

```
{
  "command": "signal",
  "property": {
    "name": <name>,
    "pid": <processid>,
    "signal": <signal>,
    "children": True
  }
}
```

To send a signal to a process child:

```
{
  "command": "signal",
  "property": {
    "name": <name>,
    "pid": <processid>,
    "signal": <signal>,
    "child_pid": <childpid>,
  }
}
```

It is also possible to send a signal to all the processes of the watcher and its childs:

```
{
  "command": "signal",
  "property": {
    "name": <name>,
  }
}
```

```
    "signal": <signal>,
    "children": True
}
```

### Command line

```
$ circusctl signal <name> [<process>] [<pid>] [--children] <signal>
```

#### Options:

- <name>: the name of the watcher
- <pid>: integer, the process id.
- <signal>: the signal number to send.
- <childpid>: the pid of a child, if any
- <children>: boolean, send the signal to all the children

Allowed signals are:

- 3: QUIT
- 15: TERM
- 9: KILL
- 1: HUP
- 21: TTIN
- 22: TTOU
- 30: USR1
- 31: USR2

### Start the arbiter or a watcher

This command starts all the processes in a watcher or all watchers.

#### ZMQ Message

```
{
  "command": "start",
  "properties": {
    "name": '<name>',
  }
}
```

The response return the status “ok”.

If the property name is present, the watcher will be started.



## Command line

```
$ circusctl start [<name>]
```

## Options

- <name>: name of the watcher

## Get process infos

You can get at any time some statistics about your processes with the `stat` command.

## ZMQ Message

To get stats for all watchers:

```
{
  "command": "stats"
}
```

To get stats for a watcher:

```
{
  "command": "stats",
  "properties": {
    "name": <name>
  }
}
```

To get stats for a process:

```
{
  "command": "stats",
  "properties": {
    "name": <name>,
    "process": <processid>
  }
}
```

The response returns an object per process with the property “info” containing some process informations:

```
{
  "info": {
    "children": [],
    "cmdline": "python",
    "cpu": 0.1,
    "ctime": "0:00.41",
    "mem": 0.1,
    "mem_info1": "3M",
    "mem_info2": "2G",
    "nice": 0,
    "pid": 47864,
    "username": "root"
  },
  "process": 5,
  "status": "ok",
}
```

```
"time": 1332265655.897085
}
```

### Command Line

```
$ circusctl stats [<watchername>] [<processid>]
```

### Get the status of a watcher or all watchers

This command start get the status of a watcher or all watchers.

### ZMQ Message

```
{
  "command": "status",
  "properties": {
    "name": '<name>',
  }
}
```

The response return the status “active” or “stopped” or the status / watchers.

### Command line

```
$ circusctl status [<name>]
```

### Options

- <name>: name of the watcher

### Example

```
$ circusctl status dummy
active
$ circusctl status
dummy: active
dummy2: active
refuge: active
```

### Stop the arbiter or a watcher

This command stop all the process in a watcher or all watchers.

### ZMQ Message

```
{
  "command": "stop",
  "properties": {
    "name": '<name>',
  }
}
```

The response return the status “ok”.

If the property name is present, then the reload will be applied to the watcher.

### Command line

```
$ circusctl stop [<name>]
```

### Options

- `<name>`: name of the watcher

## 5.6 The Web Console

Circus comes with a Web Console that can be used to manage the system.

The Web Console lets you:

- Connect to any running Circus system
- Watch the processes CPU and Memory usage in real-time
- Add or kill processes
- Add new watchers

---

**Note:** The real-time CPU & Memory usage feature uses the stats socket. If you want to activate it, make sure the Circus system you’ll connect to has the stats endpoint enabled in its configuration:

```
[circus]
...
stats_endpoint = tcp://127.0.0.1:5557
...
```

By default, this option is not activated.

---

The web console needs a few dependencies that you can install them using the `web-requirements.txt` file. Additionally, you will need to have `gevent` (and thus `libevent`) installed on your system to make this working:

```
$ bin/pip install -r web-requirements.txt
```

To enable the console, add a few options in the Circus ini file:

```
[circus]
httpd = True
httpd_host = localhost
httpd_port = 8080
```

*htpd\_host* and *htpd\_port* are optional, and default to *localhost* and *8080*.

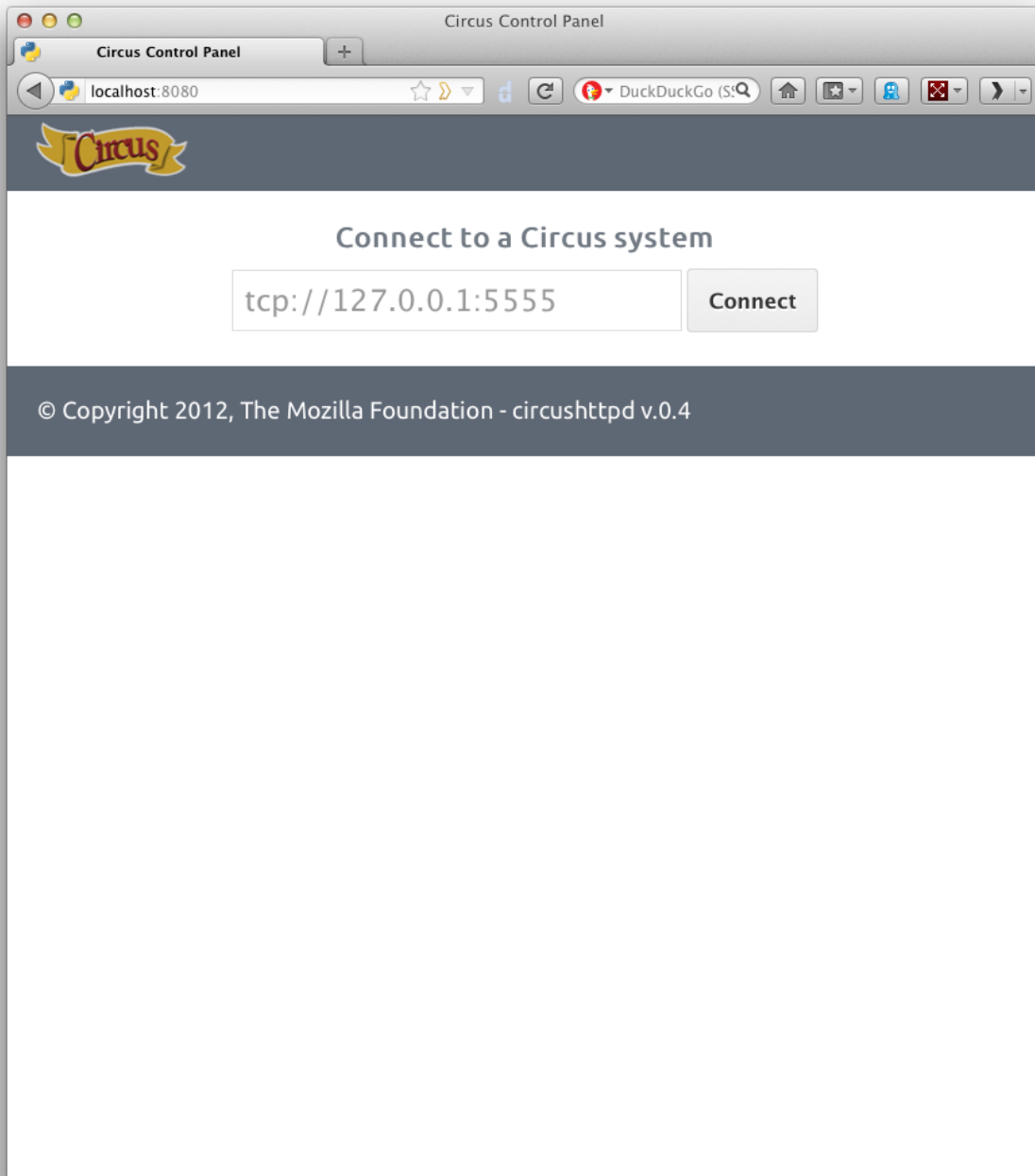
If you want to run the web app on its own, just run the **circushtpd** script:

```
$ circushtpd
Bottle server starting up...
Listening on http://localhost:8080/
Hit Ctrl-C to quit.
```

By default the script will run the Web Console on port 8080, but the `-port` option can be used to change it.

### 5.6.1 Using the console

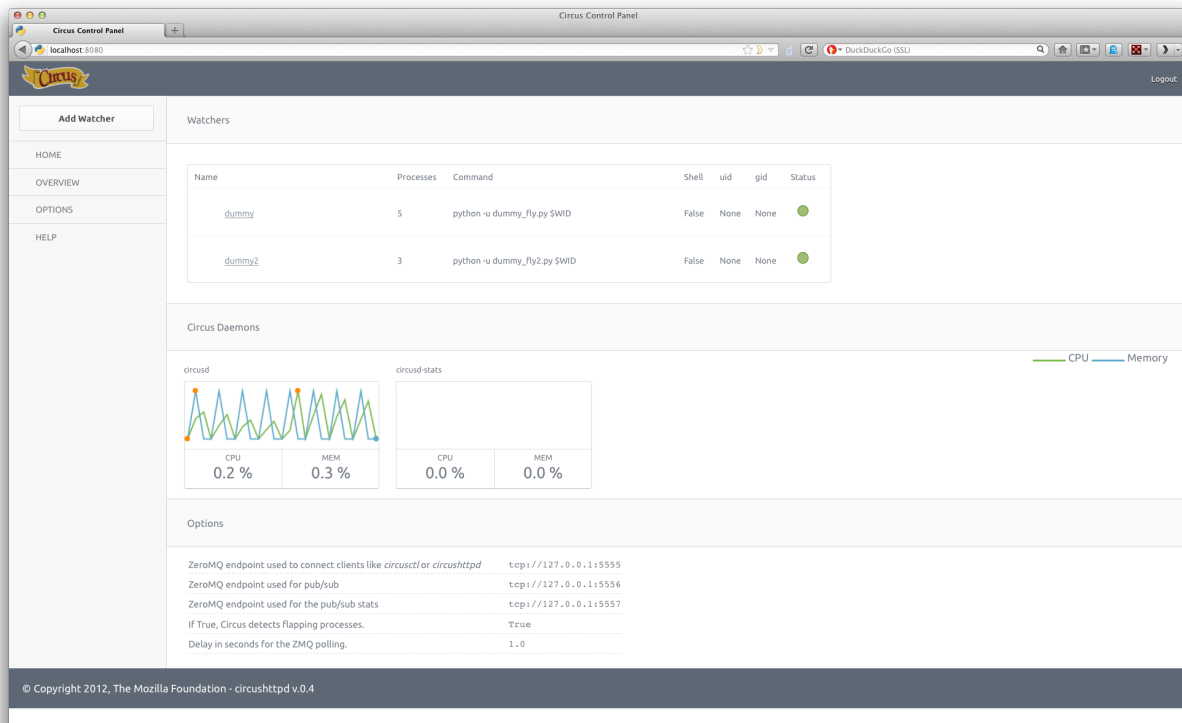
Once the script is running, you can open a browser and visit *http://localhost:8080*. You should get this screen:



The Web Console is ready to be connected to a Circus system, given its **endpoint**. By default the endpoint is *tcp://127.0.0.1:5555*.

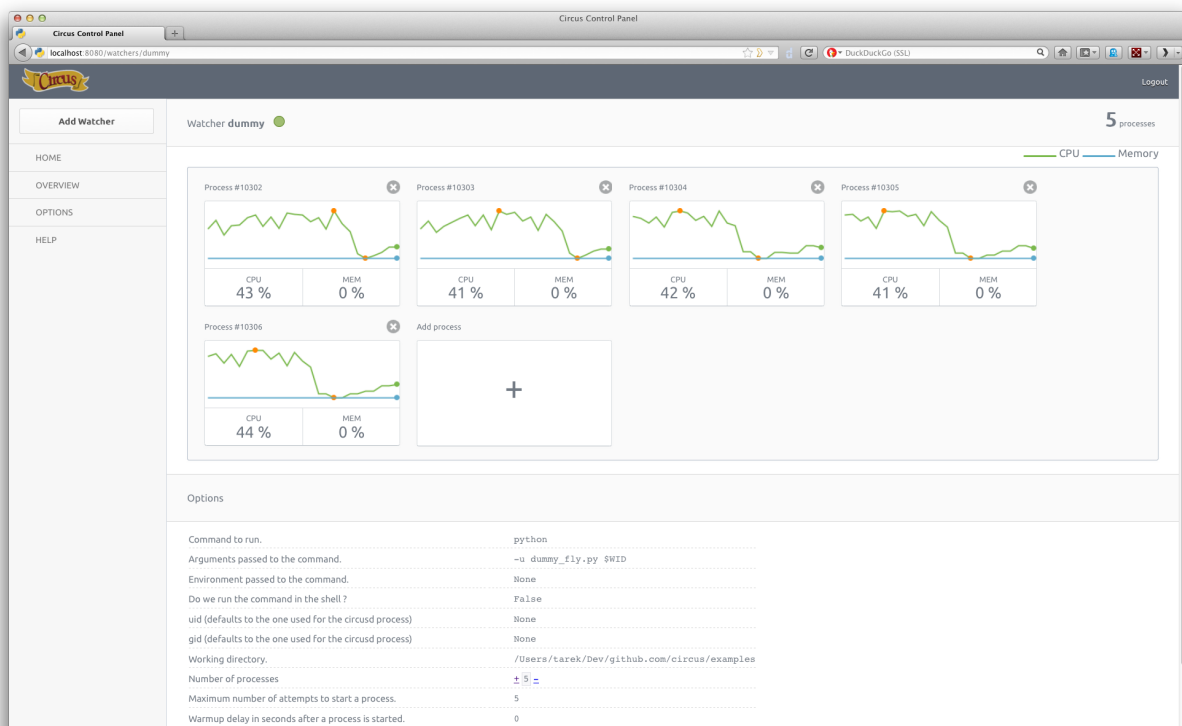
Once you hit *Connect*, the web application will connect to the Circus system.

With the Web Console logged in, you should get a list of watchers, and a real-time status of the two Circus processes (circusd and circusd-stats).



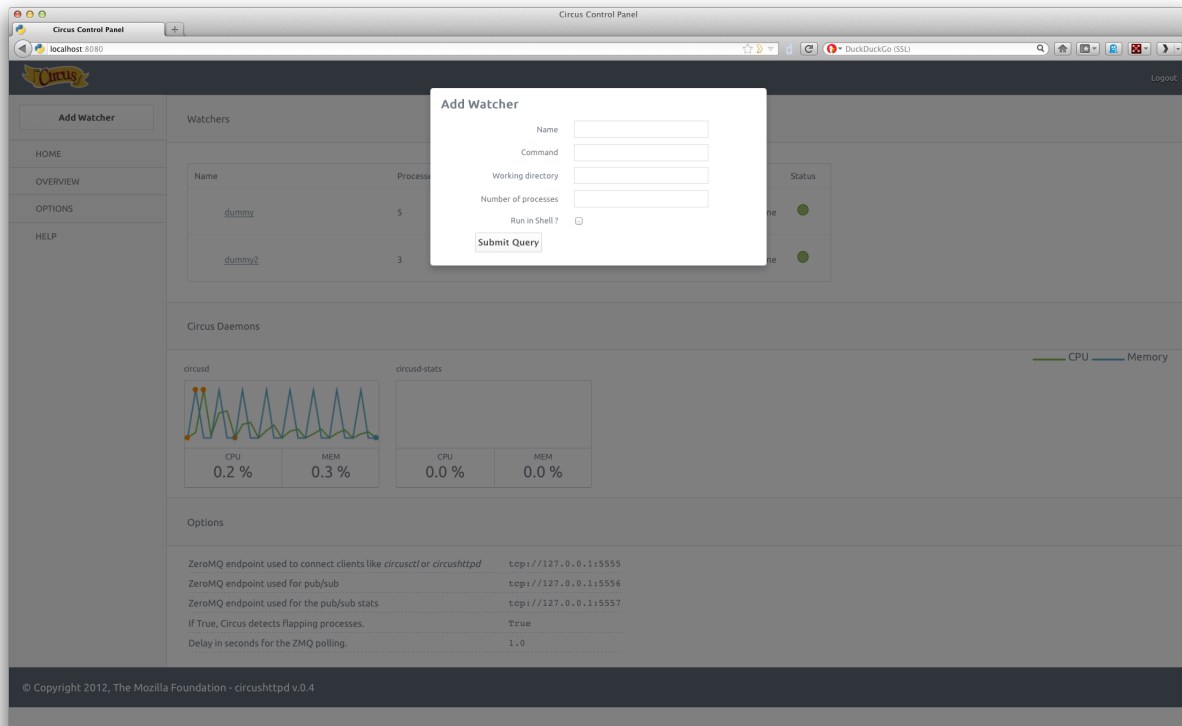
You can click on the status of each watcher to toggle it from **Active** (green) to **Inactive** (red). This change is effective immediately and let you start & stop watchers.

If you click on the watcher name, you will get a web page for that particular watcher, with its processes:



On this screen, you can add or remove processes, and kill existing ones.

Last but not least, you can add a brand new watcher by clicking on the *Add Watcher* link in the left menu:



## 5.6.2 Embedding circushttd into Circus

*circushttd* is a WSGI application so you can run it with any web server that's compatible with that protocol. By default it uses the standard library **wsgiref** server, but that server does not really support any load.

You can use **Chaussette** to bind a WSGI server and have *circushttd* managed by Circus itself.

To do so, make sure Chaussette is installed:

```
$ pip install chaussette
```

Then add a new *watcher* and a *socket* sections in your ini file:

```
[watcher:webconsole]
cmd = chaussette --fd $(circus.sockets.webconsole) circus.web.circushttd.app
singleton = 1
use_sockets = 1

[socket:webconsole]
host = 127.0.0.0
port = 8080
```

That's it !

## 5.6.3 Running behind Nginx

Nginx can act as a proxy in front of Circus. It can also deal with security.

To hook Nginx, you define a *location* directive that proxies the calls to Circus.

Example:

```
location ~/media/*(.jpg|.css|.js)$ {
    alias /path/to/circus/web/;
}

location / {
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header Host $http_host;
    proxy_redirect off;
    proxy_pass http://127.0.0.1:8080;
}
```

If you want more configuration options, see <http://wiki.nginx.org/HttpProxyModule>.

### 5.6.4 Password-protect circushttpd

As explained in the *Security* page, running *circushttpd* is pretty unsafe. We don't provide any security in Circus itself, but you can protect your console at the NGinx level, by using <http://wiki.nginx.org/HttpAuthBasicModule>

Example:

```
location / {
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header Host $http_host;
    proxy_redirect off;
    proxy_pass http://127.0.0.1:8080;
    auth_basic "Restricted";
    auth_basic_user_file /path/to/htpasswd;
}
```

The **htpasswd** file contains users and their passwords, and a password prompt will pop when you access the console.

You can use Apache's `htpasswd` script to edit it, or the Python script they provide at: <http://trac.edgewall.org/browser/trunk/contrib/htpasswd.py>

Of course that's just one way to protect your web console, you could use many other techniques.

### 5.6.5 Extending the web console

We picked *bottle* to build the webconsole, mainly because it's a really tiny framework that doesn't do much. By having a look at the code of the web console, you'll eventually find out that it's really simple to understand.

Here is how it's split:

- The *circushttpd.py* file contains the “views” definitions and some code to handle the socket connection (via `socketio`).
- the *controller.py* contains a single class which is in charge of doing the communication with the circus controller. It allows to have a nicer high level API when defining the web server.

If you want to add a feature in the web console you can reuse the code that's existing. A few tools are at your disposal to ease the process:

- There is a *render\_template* function, which takes the named arguments you pass to it and pass them to the template renderer and return the resulting HTML. It also passes some additional variables, such as the session, the circus version and the client if defined.



- If you want to run commands and do redirection depending the result of it, you can use the `run_command` function, which takes a callable as a first argument, a message in case of success and a redirection url.

The `StatsNamespace` class is responsible for managing the websocket communication on the server side. Its documentation should help you to understand what it does.

## 5.7 Circus Sockets

Circus can bind network sockets and manage them as it does for processes.

The main idea is that a child process that's created by Circus to run one of the watcher's command can inherit from all the opened file descriptors.

That's how Apache or Unicorn works, and many other tools out there.

### 5.7.1 Goal

The goal of having sockets managed by Circus is to be able to manage network applications in Circus exactly like other applications.

For example, if you use Circus with [Chaussette](#) – a WSGI server, you can get a very fast web server running and manage “*Web Workers*” in Circus as you would do for any other process.

Splitting the socket management from the network application itself offers a lot of opportunities to scale and manage your stack.

### 5.7.2 Design

The gist of the feature is done by binding the socket and start listening to it in **circusd**:

```
import socket

sock = socket.socket(FAMILY, TYPE)
sock.bind((HOST, PORT))
sock.listen(BACKLOG)
fd = sock.fileno()
```

Circus then keeps track of all the opened fds, and let the processes it runs as children have access to them if they want.

If you create a small Python network script that you intend to run in Circus, it could look like this:

```
import socket
import sys

fd = int(sys.argv[1]) # getting the FD from circus
sock = socket.fromfd(fd, FAMILY, TYPE)

# dealing with one request at a time
while True:
    conn, addr = sock.accept()
    request = conn.recv(1024)
    .. do something ..
    conn.sendall(response)
    conn.close()
```

Then Circus could run like this:

```
[circus]
check_delay = 5
endpoint = tcp://127.0.0.1:5555
pubsub_endpoint = tcp://127.0.0.1:5556
stats_endpoint = tcp://127.0.0.1:5557

[watcher:dummy]
cmd = mycoolscript $(circus.sockets.foo)
use_sockets = True
warmup_delay = 0
numprocesses = 5

[socket:foo]
host = 127.0.0.1
port = 8888
```

`$(circus.sockets.foo)` will be replaced by the FD value once the socket is created and bound on the 8888 *port*.

### 5.7.3 Real-world example

**Chaussette** is the perfect Circus companion if you want to run your WSGI application.

Once it's installed, running 5 **meinheld** workers can be done by creating a socket and calling the **chaussette** command in a worker, like this:

```
[circus]
endpoint = tcp://127.0.0.1:5555
pubsub_endpoint = tcp://127.0.0.1:5556
stats_endpoint = tcp://127.0.0.1:5557

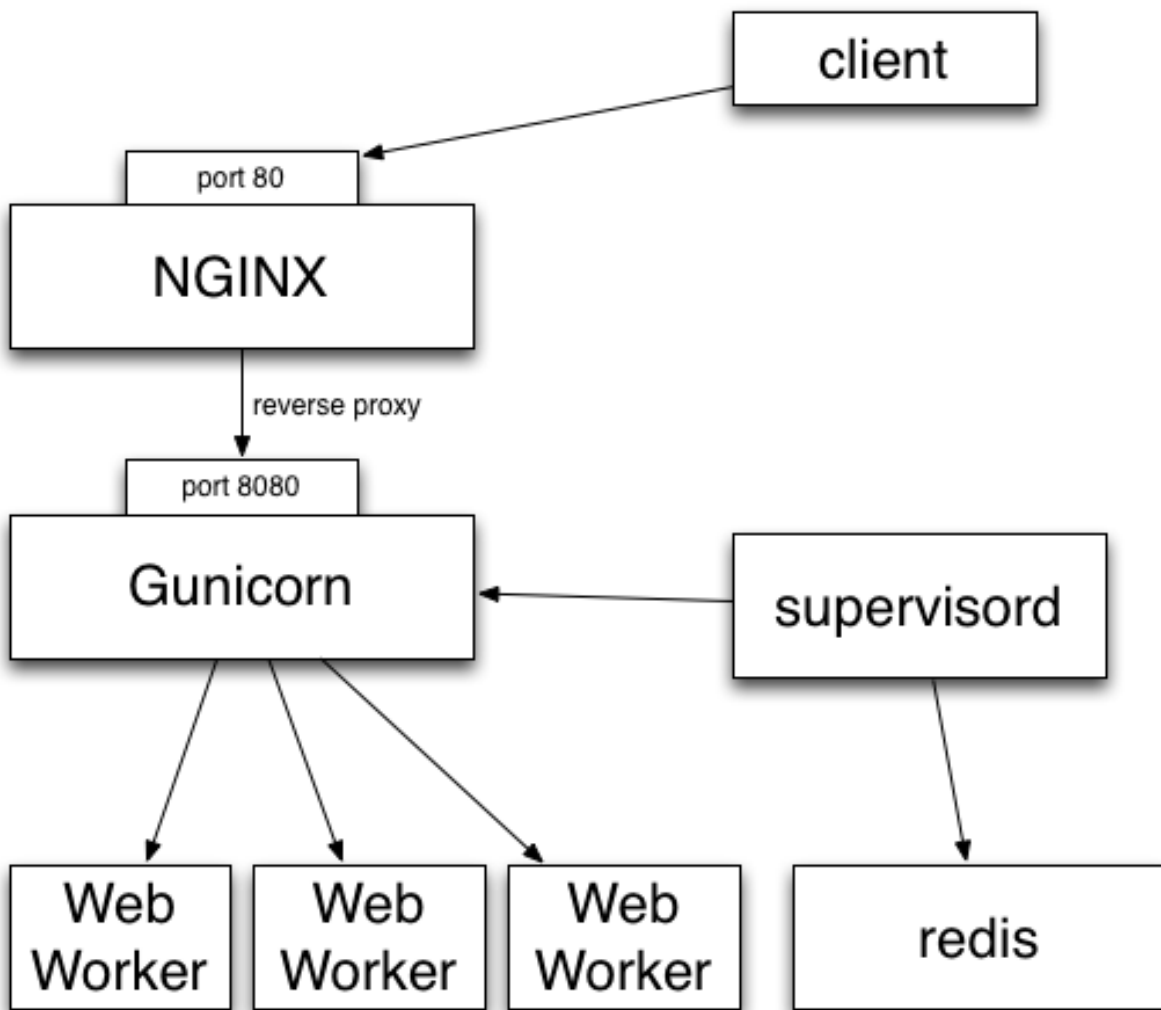
[watcher:web]
cmd = chaussette --fd $(circus.sockets.web) --backend meinheld mycool.app
use_sockets = True
numprocesses = 5

[socket:web]
host = 0.0.0.0
port = 8000
```

We did not publish benchmarks yet, but a Web cluster managed by Circus with a Gevent or Meinheld backend is as fast as any pre-fork WSGI server out there.

### 5.7.4 Circus stack v.s. Classical stack

In a classical WSGI stack, you have a server like Gunicorn that serves on a port or an unix socket and is usually deployed behind a web server like Nginx:



Clients call Nginx, which reverse proxies all the calls to Gunicorn.

If you want to make sure the Gunicorn process stays up and running, you have to use a program like Supervisord or upstart.

Gunicorn in turn watches for its processes (“workers”).

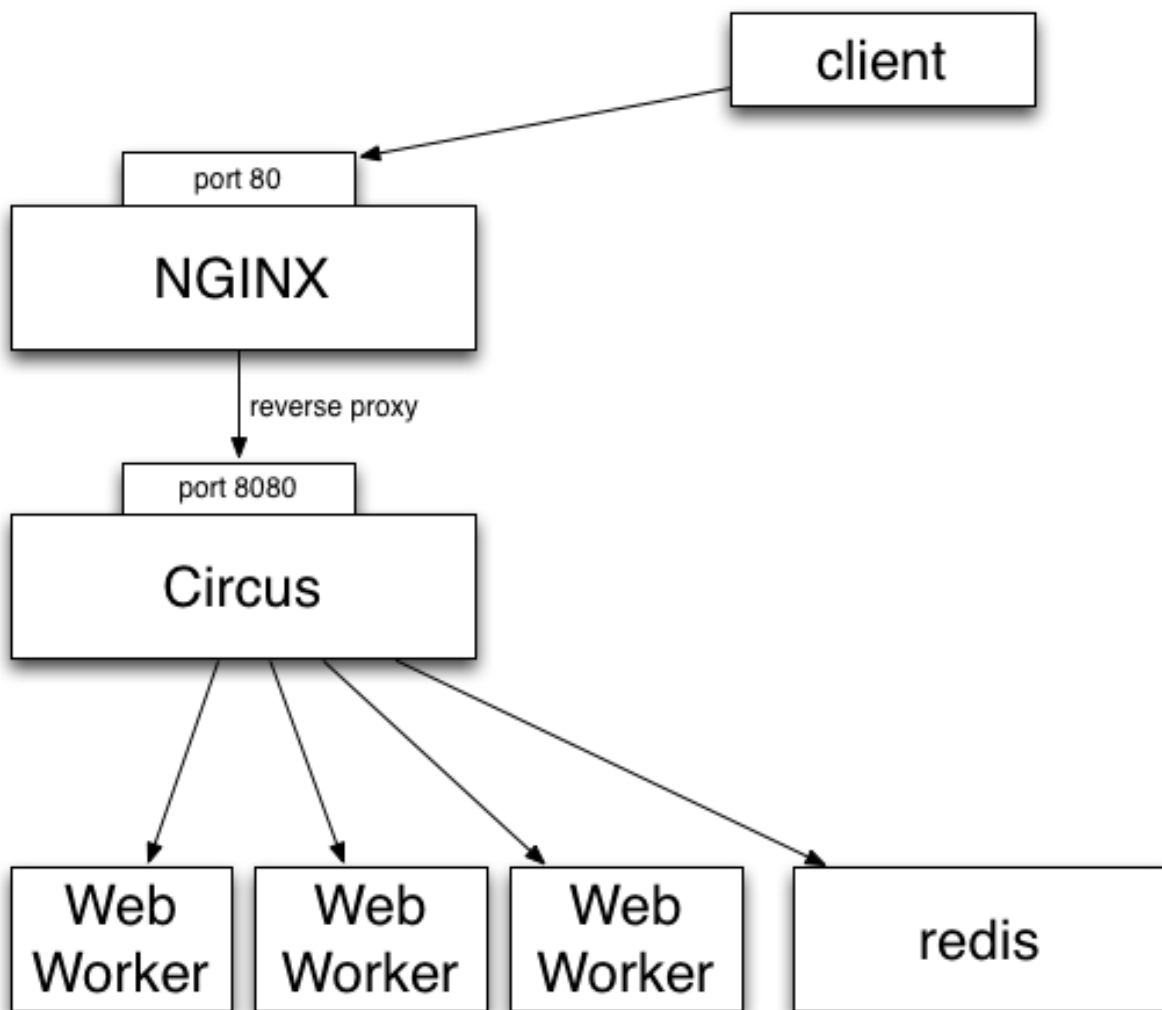
In other words you are using two levels of process management. One that you manage and control (supervisord), and a second one that you have to manage in a different UI, with a different philosophy and less control over what’s going on (the wsgi server’s one)

This is true for Gunicorn and most multi-processes WSGI servers out there I know about. uWsgi is a bit different as it offers plethoras of options.

But if you want to add a Redis server in your stack, you *will* end up with managing your stack processes in two different places.

Circus’ approach on this is to manage processes *and* sockets.

A Circus stack can look like this:



So, like Gunicorn, Circus is able to bind a socket that will be proxied by Nginx. Circus don't deal with the requests but simply binds the socket. It's then up to a web worker process to accept connections on the socket and do the work.

It provides equivalent features than Supervisor but will also let you manage all processes at the same level, whether they are web workers or Redis or whatever. Adding a new web worker is done exactly like adding a new Redis process.

## 5.8 Benches

We did a few benches to compare Circus & Chaussette with Gunicorn. To summarize, Circus is not adding any overhead and you can pick up many different backends for your web workers.

See:

- <http://blog.ziade.org/2012/06/28/wsgi-web-servers-bench>
- <http://blog.ziade.org/2012/07/03/wsgi-web-servers-bench-part-2>

## 5.9 Circus Library

The Circus package is composed of a high-level `get_arbiter()` function and many classes. In most cases, using the high-level function should be enough, as it creates everything that is needed for Circus to run.

You can subclass Circus' classes if you need more granularity than what is offered by the configuration.

### 5.9.1 The `get_arbiter` function

`get_arbiter()` is just a convenience on top of the various circus classes. It creates a *arbiter* (class `Arbiter`) instance with the provided options, which in turn runs a single `Watcher` with a single `Process`.

```
circus.get_arbiter(watchers, controller='tcp://127.0.0.1:5555', pub-
                   sub_endpoint='tcp://127.0.0.1:5556', stats_endpoint=None, env=None,
                   name=None, context=None, background=False, stream_backend='thread',
                   plugins=None)
```

Creates a `Arbiter` and a single watcher in it.

Options:

- **watchers** – a list of watchers. A watcher in that case is a dict containing:

- **name** – the name of the watcher (default: `None`)

- **cmd** – the command line used to run the `Watcher`.

- **args** – the args for the command (list or string).

- **executable** – When `executable` is given, the first item in the `args` sequence obtained from **cmd** is still treated by most programs as the command name, which can then be different from the actual executable name. It becomes the display name for the executing program in utilities such as **ps**.

- **numprocesses** – the number of processes to spawn (default: 1).

- **warmup\_delay** – the delay in seconds between two spawns (default: 0)

- **shell** – if `True`, the processes are run in the shell (default: `False`)

- **working\_dir** – the working dir for the processes (default: `None`)

- **uid** – the user id used to run the processes (default: `None`)

- **gid** – the group id used to run the processes (default: `None`)

- **env** – the environment passed to the processes (default: `None`)

- **send\_hup**: if `True`, a process reload will be done by sending the `SIGHUP` signal. (default: `False`)

- **stdout\_stream**: a mapping containing the options for configuring the stdout stream. Default to `None`. When provided, may contain:

- \***class**: the fully qualified name of the class to use for streaming. Defaults to `circus.stream.FileStream`

- \***refresh\_time**: the delay between two stream checks. Defaults to 0.3 seconds.

- \*any other key will be passed the class constructor.

- **stderr\_stream**: a mapping containing the options for configuring the stderr stream. Default to `None`. When provided, may contain:

- \***class**: the fully qualified name of the class to use for streaming. Defaults to `circus.stream.FileStream`

\***refresh\_time**: the delay between two stream checks. Defaults to 0.3 seconds.

\*any other key will be passed the class constructor.

–**max\_retry**: the number of times we attempt to start a process, before we abandon and stop the whole watcher. (default: 5)

•**controller** – the zmq entry point (default: ‘tcp://127.0.0.1:5555’)

•**pubsub\_endpoint** – the zmq entry point for the pubsub (default: ‘tcp://127.0.0.1:5556’)

•**stats\_endpoint** – the stats endpoint. If not provided, the *circusd-stats* process will not be launched. (default: None)

•**context** – the zmq context (default: None)

•**background** – If True, the arbiter is launched in a thread in the background (default: False)

•**stream\_backend** – the backend that will be used for the streaming process. Can be *thread* or *gevent*. When set to *gevent* you need to have *gevent* and *gevent\_zmq* installed. (default: thread)

•**plugins** – a list of plugins. Each item is a mapping with:

–**use** – Fully qualified name that points to the plugin class

–every other value is passed to the plugin in the **config** option

Example:

```
from circus import get_arbiter

arbiter = get_arbiter("myprogram", numprocesses=3)
try:
    arbiter.start()
finally:
    arbiter.stop()
```

## 5.9.2 The classes collection

Circus provides a series of classes you can use to implement your own process manager:

- **Process**: wraps a running process and provides a few helpers on top of it.
- **Watcher**: run several instances of **Process** against the same command. Manage the death and life of processes.
- **Arbiter**: manages several **Watcher**.

```
class circus.process.Process(wid, cmd, args=None, working_dir=None, shell=False,
                             uid=None, gid=None, env=None, rlimits=None, executable=None,
                             use_fds=False, watcher=None, spawn=True)
```

Wraps a process.

Options:

- wid**: the process unique identifier. This value will be used to replace the *\$WID* string in the command line if present.
- cmd**: the command to run. May contain any of the variables available that are being passed to this class. They will be replaced using the python format syntax.
- 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.

- executable**: When executable is given, the first item in the args sequence obtained from **cmd** is still treated by most programs as the command name, which can then be different from the actual executable name. It becomes the display name for the executing program in utilities such as **ps**.
- working\_dir**: the working directory to run the command in. If not provided, will default to the current working directory.
- shell**: if *True*, will run the command in the shell environment. *False* by default. **warning: this is a security hazard.**
- uid**: if given, is the user id or name the command should run with. The current uid is the default.
- gid**: if given, is the group id or name the command should run with. The current gid is the default.
- env**: a mapping containing the environment variables the command will run with. Optional.
- rlimits**: a mapping containing rlimit names and values that will be set before the command runs.
- use\_fds** XXX

**pid**

Return the *pid*

**stdout**

Return the *stdout* stream

**stderr**

Return the *stdout* stream

**send\_signal** (\*args, \*\*kw)

Sends a signal **sig** to the process.

**stop** (\*args, \*\*kw)

Terminate the process.

**age** ()

Return the age of the process in seconds.

**info** ()

Return process info.

The info returned is a mapping with these keys:

- mem\_info1**: Resident Set Size Memory in bytes (RSS)
- mem\_info2**: Virtual Memory Size in bytes (VMS).
- cpu**: % of cpu usage.
- mem**: % of memory usage.
- ctime**: process CPU (user + system) time in seconds.
- pid**: process id.
- username**: user name that owns the process.
- nice**: process niceness (between -20 and 20)
- cmdline**: the command line the process was run with.

**children** ()

Return a list of children pids.

**is\_child** (pid)

Return True is the given *pid* is a child of that process.

**send\_signal\_child** (\*args, \*\*kw)

Send signal *signum* to child *pid*.

**send\_signal\_children** (\*args, \*\*kw)

Send signal *signum* to all children.

**status**

Return the process status as a constant

- RUNNING
- DEAD\_OR\_ZOMBIE
- UNEXISTING
- OTHER

Example:

```
>>> from circus.process import Process
>>> process = Process('Top', 'top', shell=True)
>>> process.age()
3.0107998847961426
>>> process.info()
'Top: 6812 N/A tarek Zombie N/A N/A N/A N/A N/A'
>>> process.status
1
>>> process.stop()
>>> process.status
2
>>> process.info()
'No such process (stopped?)'
```

**class** circus.watcher.**Watcher** (*name*, *cmd*, *args*=None, *numprocesses*=1, *warmup\_delay*=0.0, *working\_dir*=None, *shell*=False, *uid*=None, *max\_retry*=5, *gid*=None, *send\_hup*=False, *env*=None, *stopped*=True, *graceful\_timeout*=30.0, *prereload\_fn*=None, *rlimits*=None, *executable*=None, *stdout\_stream*=None, *stderr\_stream*=None, *stream\_backend*='thread', *priority*=0, *singleton*=False, *use\_sockets*=False, *copy\_env*=False, \*\**options*)

Class managing a list of processes for a given command.

Options:

- name**: name given to the watcher. Used to uniquely identify it.
- cmd**: the command to run. May contain *\$WID*, which will be replaced by **wid**.
- 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.
- numprocesses**: Number of processes to run.
- working\_dir**: the working directory to run the command in. If not provided, will default to the current working directory.
- shell**: if *True*, will run the command in the shell environment. *False* by default. **warning: this is a security hazard.**
- uid**: if given, is the user id or name the command should run with. The current uid is the default.
- gid**: if given, is the group id or name the command should run with. The current gid is the default.
- send\_hup**: if *True*, a process reload will be done by sending the SIGHUP signal. Defaults to *False*.



- env**: a mapping containing the environment variables the command will run with. Optional.
- rlimits**: a mapping containing rlimit names and values that will be set before the command runs.
- stdout\_stream**: a mapping that defines the stream for the process stdout. Defaults to None.

Optional. When provided, *stdout\_stream* is a mapping containing up to three keys:

- class**: the stream class. Defaults to *circus.stream.FileStream*
- filename**: the filename, if using a *FileStream*
- refresh\_time**: the delay between two stream checks. Defaults to 0.3 seconds.

This mapping will be used to create a stream callable of the specified class. Each entry received by the callable is a mapping containing:

- pid** - the process pid
- name** - the stream name (*stderr* or *stdout*)
- data** - the data

- stderr\_stream**: a mapping that defines the stream for the process stderr. Defaults to None.

Optional. When provided, *stderr\_stream* is a mapping containing up to three keys: - **class**: the stream class. Defaults to *circus.stream.FileStream* - **filename**: the filename, if using a *FileStream* - **refresh\_time**: the delay between two stream checks. Defaults

to 0.3 seconds.

This mapping will be used to create a stream callable of the specified class.

Each entry received by the callable is a mapping containing:

- pid** - the process pid
- name** - the stream name (*stderr* or *stdout*)
- data** - the data

- stream\_backend** – the backend that will be used for the streaming process. Can be *thread* or *gevent*. When set to *gevent* you need to have *gevent* and *gevent\_zmq* installed. (default: *thread*)
- priority** – integer that defines a priority for the watcher. When the Arbiter do some operations on all watchers, it will sort them with this field, from the bigger number to the smallest. (default: 0)
- singleton** – If True, this watcher has a single process. (default:False)

= **use\_sockets** – XXX

- copy\_env** – If True, the environment in which circus had been run will be reproduced for the workers.
- options** – extra options for the worker. All options found in the configuration file for instance, are passed in this mapping – this can be used by plugins for watcher-specific options.

**notify\_event** (*topic, msg*)

Publish a message on the event publisher channel

**reap\_processes** (*\*args, \*\*kw*)

Reap all the processes for this watcher.

**manage\_processes** (*\*args, \*\*kw*)

manage processes

**reap\_and\_manage\_processes** (*\*args, \*\*kw*)

Reap & manage processes.

**spawn\_processes** (\*args, \*\*kw)

Spawn processes.

**spawn\_process** ()

Spawn process.

**kill\_process** (process, sig=15)

Kill process.

**kill\_processes** (\*args, \*\*kw)

Kill all the processes of this watcher.

**send\_signal\_child** (\*args, \*\*kw)

Send signal to a child.

**stop** (\*args, \*\*kw)

Stop.

**start** (\*args, \*\*kw)

Start.

**restart** (\*args, \*\*kw)

Restart.

**reload** (\*args, \*\*kw)

**class** circus.arbiter.**Arbiter** (watchers, endpoint, pubsub\_endpoint, check\_delay=1.0, pre-reload\_fn=None, context=None, loop=None, stats\_endpoint=None, plugins=None, sockets=None, warmup\_delay=0, httpd=False, httpd\_host='localhost', httpd\_port=8080)

Class used to control a list of watchers.

Options:

- **watchers** – a list of `Watcher` objects
- **endpoint** – the controller ZMQ endpoint
- **pubsub\_endpoint** – the pubsub endpoint
- **stats\_endpoint** – the stats endpoint. If not provided, the *circusd-stats* process will not be launched.
- **check\_delay** – the delay between two controller points (default: 1 s)
- **prereload\_fn** – callable that will be executed on each reload (default: None)
- **context** – if provided, the zmq context to reuse. (default: None)
- **loop: if provided, a `zmq.eventloop.ioloop.IOLoop` instance** to reuse. (default: None)
- **plugins** – a list of plugins. Each item is a mapping with:
  - **use** – Fully qualified name that points to the plugin class
  - every other value is passed to the plugin in the **config** option
- **sockets** – a mapping of sockets. Each key is the socket name, and each value a `CircusSocket` class. (default: None)
- **warmup\_delay** – a delay in seconds between two watchers startup. (default: 0)
- **httpd** – If True, a *circushttpd* process is run (default: False)
- **httpd\_host** – the *circushttpd* host (default: localhost)
- **httpd\_port** – the *circushttpd* port (default: 8080)

**start** (\*args, \*\*kw)

Starts all the watchers.

The start command is an infinite loop that waits for any command from a client and that watches all the processes and restarts them if needed.

**reload** (\*args, \*\*kw)

Reloads everything.

Run the `prereload_fn()` callable if any, then gracefully reload all watchers.

**numprocesses** ()

Return the number of processes running across all watchers.

**numwatchers** ()

Return the number of watchers.

**get\_watcher** (name)

Return the watcher *name*.

**add\_watcher** (name, cmd, \*\*kw)

Adds a watcher.

Options:

- **name**: name of the watcher to add
- **cmd**: command to run.
- all other options defined in the `Watcher` constructor.

## 5.10 The Plugin System

Circus comes with a plugin system which let you interact with **circusd**.

---

**Note:** We might add `circusd-stats` support to plugins later on

---

A Plugin is composed of two parts:

- a ZMQ subscriber to all events published by **circusd**
- a ZMQ client to send commands to **circusd**

Each plugin is run as a separate process under a custom watcher.

A few examples of some plugins you could create with this system:

- a notification system that sends e-mail alerts when a watcher is flapping
- a logger
- a tool that add or remove processes depending on the load
- etc.

Circus itself provides a few plugins:

- a `statsd` plugin, that sends to `statsd` all events emitted by `circusd`
- the flapping feature which avoid to re-launch processes infinitely when they die too quickly.
- many more to come !

### 5.10.1 The CircusPlugin class

Circus provides a base class to help you implement plugins: `circus.plugins.CircusPlugin`

**class** `circus.plugins.CircusPlugin` (*endpoint, pubsub\_endpoint, check\_delay, \*\*config*)  
Base class to write plugins.

Options:

- context** – the ZMQ context to use
- endpoint** – the circusd ZMQ endpoint
- pubsub\_endpoint** – the circusd ZMQ pub/sub endpoint
- check\_delay** – the configured check delay
- config** – free config mapping

**call** (*command, \*\*props*)  
Sends to **circusd** the command.

Options:

- command** – the command to call
- props** – keywords argument to add to the call

Returns the JSON mapping sent back by **circusd**

**cast** (*command, \*\*props*)  
Fire-and-forget a command to **circusd**

Options:

- command** – the command to call
- props** – keywords argument to add to the call

**handle\_recv** (*data*)  
Receives every event published by **circusd**

Options:

- data** – a tuple containing the topic and the message.

**handle\_stop** ()  
Called right before the plugin is stopped by Circus.

**handle\_init** ()  
Called right before a plugin is started - in the thread context.

When initialized by Circus, this class creates its own event loop that receives all **circusd** events and pass them to `handle_recv()`. The data received is a tuple containing the topic and the data itself.

`handle_recv()` **must** be implemented by the plugin.

The `call()` and `cast()` methods can be used to interact with **circusd** if you are building a Plugin that actively interacts with the daemon.

`handle_init()` and `handle_stop()` are just convenience methods you can use to initialize and clean up your code. `handle_init()` is called within the thread that just started. `handle_stop()` is called in the main thread just before the thread is stopped and joined.

### 5.10.2 Writing a plugin

Let's write a plugin that logs in a file every event happening in **circusd**. It takes one argument which is the filename.

The plugin could look like this:

```
from circus.plugins import CircusPlugin

class Logger(CircusPlugin):

    name = 'logger'

    def __init__(self, filename, **kwargs):
        super(Logger, self).__init__(**kwargs)
        self.filename = filename
        self.file = None

    def handle_init(self):
        self.file = open(self.filename, 'a+')

    def handle_stop(self):
        self.file.close()

    def handle_recv(self, data):
        topic, msg = data
        self.file.write('%s::%s' % (topic, msg))
```

That's it ! This class can be saved in any package/module, as long as it can be seen by Python.

For example, `Logger` could be found in a *plugins* module in a *myproject* package.

### 5.10.3 Using a plugin

You can run a plugin through the command line with the **circus-plugin** command, by specifying the plugin fully qualified name:

```
$ circus-plugin --endpoint tcp://127.0.0.1:5555 --pubsub tcp://127.0.0.1:5556 myproject.plugins.Logger
[INFO] Loading the plugin...
[INFO] Endpoint: 'tcp://127.0.0.1:5555'
[INFO] Pub/sub: 'tcp://127.0.0.1:5556'
[INFO] Starting
```

Another way to run a plugin is to let Circus handle its initialization. This is done by adding a **[plugin:NAME]** section in the configuration file, where *NAME* is a unique name for your plugin:

```
[plugin:logger]
use = myproject.plugins.Logger
filename = /var/myproject/circus.log
```

**use** is mandatory and points to the fully qualified name of the plugin.

When Circus starts, it creates a watcher with one process that runs the pointed class, and pass any other variable contained in the section to the plugin constructor via the **config** mapping.

You can also programmatically add plugins when you create a `circus.arbiter.Arbiter` class or use `circus.get_arbiter()`, see *Circus Library*.

### 5.10.4 Performances

Since every plugin is loaded in its own process, it should not impact the overall performances of the system as long as the work done by the plugin is not doing too many calls to the **circusd** process.

## 5.11 Deployment

This section will contain recipes to deploy Circus. Until then you can look at Pete's Puppet recipe at <https://github.com/fetep/puppet-circus>

This Puppet recipe also contains an *upstart* script you can reuse.

## 5.12 Security

Circus is built on the top of the ZeroMQ library and comes with no security at all.

There were no focus yet on protecting the Circus system from attacks on its ports, and depending on how you run it, you are creating a potential security hole in the system.

This section explains what Circus does on your system when you run it, and a few recommendations if you want to protect your server.

You can also read <http://www.zeromq.org/area:faq#toc5>

### 5.12.1 TCP ports

By default, Circus opens the following TCP ports on the local host:

- **5555** – the port used to control circus via **circusctl**
- **5556** – the port used for the Publisher/Subscriber channel.

These ports allow client apps to interact with your Circus system, and depending on how your infrastructure is organized, you may want to protect these ports via firewalls **or** to configure Circus to run using **IPC** ports. When Configured using IPC, the commands must be run from the same box, but no one can access them from outside unlike TCP.

### 5.12.2 uid and gid

By default, all processes started with Circus will be running with the same user and group than **circusd**. Depending on the privileges the user has on the system, you may not have access to all the features Circus provides.

For instance, some statistics features on the running processes require privileges. Typically, if the CPU usage numbers you get using the **stats** command are 0, it means your user can't access the proc files.

You may run **circusd** as root, to fix this, and set the **uid** and **gid** values for each watcher to get all features.

But beware that running **circusd** as root exposes you to potential privilege escalation bugs. While we're doing our best to avoid any bug, running as root and facing a bug that performs unwanted actions on your system may be an issue.

The best way to prevent this is to make sure that the system running Circus is isolated (like a VM) **or** to run the whole system under a controlled user.

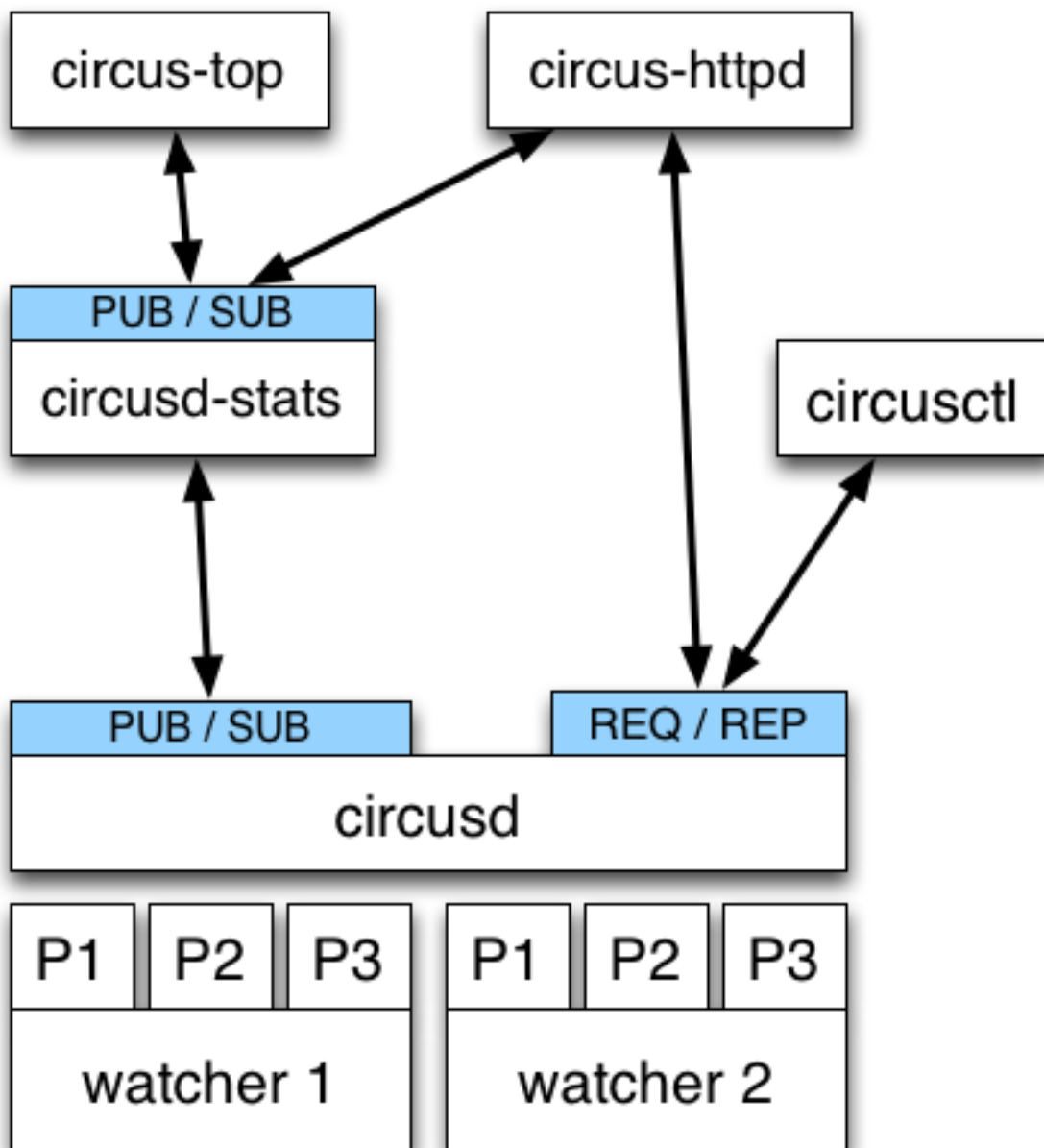
### 5.12.3 circushttpd

The web application is not secured at all and once connected on a running Circus, it can do anything and everything.

#### Do not make it publicly available

If you want to protect the access to the web panel, you can serve it behind Nginx or Apache or any proxy-capable web server, than can set up security.

## 5.13 Design



Circus is composed of a main process called **circusd** which takes care of running all the processes. Each process managed by Circus is a child process of **circusd**.

Processes are organized in groups called **watchers**. A **watcher** is basically a command **circusd** runs on your system, and for each command you can configure how many processes you want to run.

The concept of *watcher* is useful when you want to manage all the processes running the same command – like restart them, etc.

**circusd** binds two ZeroMQ sockets:

- **REQ/REP** – a socket used to control **circusd** using json-based *commands*.
- **PUB/SUB** – a socket where **circusd** publishes events, like when a process is started or stopped.

Another process called **circusd-stats** is run by **circusd** when the option is activated. **circusd-stats**'s job is to publish CPU/Memory usage statistics in a dedicated **PUB/SUB** channel.

This specialized channel is used by **circus-top** and **circus-httpd** to display a live stream of the activity.

**circus-top** is a console script that mimics **top** to display all the CPU and Memory usage of the processes managed by Circus.

**circus-httpd** is the web management interface that will let you interact with Circus. It displays a live stream using web sockets and the **circusd-stats** channel, but also let you interact with **circusd** via its **REQ/REP** channel.

Last but not least, **circusctl** is a command-line tool that let you drive **circusd** via its **REQ/REP** channel.

You can also have plugins that subscribe to **circusd**'s **PUB/SUB** channel and let you send commands to the **REQ/REP** channel like **circusctl** would.

## 5.14 Why should I use Circus instead of X ?

### 1. Circus simplifies your web stack process management

Circus knows how to manage processes *and* sockets, so you don't have to delegate web workers management to a WSGI server.

See *Circus stack v.s. Classical stack*

### 2. Circus provides pub/sub and poll notifications via ZeroMQ

Circus has a *pub/sub* channel you can subscribe to. This channel receives all events happening in Circus. For example, you can be notified when a process is *flapping*, or build a client that triggers a warning when some processes are eating all the CPU or RAM.

These events are sent via a ZeroMQ channel, which makes it different from the stdin stream Supervisor<sup>sd</sup> uses:

- Circus sends events in a fire-and-forget fashion, so there's no need to manually loop through *all* listeners and maintain their states.
- Subscribers can be located on a remote host.

Circus also provides ways to get status updates via one-time polls on a req/rep channel. This means you can get your information without having to subscribe to a stream. The *Command-line tools* command provided by Circus uses this channel.

See *Examples*.

### 3. Circus is (Python) developer friendly



While Circus can be driven entirely by a config file and the *circusctl* / *circusd* commands, it is easy to reuse all or part of the system to build your own custom process watcher in Python.

Every layer of the system is isolated, so you can reuse independently:

- the process wrapper (*Process*)
- the processes manager (*Watcher*)
- the global manager that runs several processes managers (*Arbiter*)
- and so on...

#### 4. Circus scales

One of the use cases of Circus is to manage thousands of processes without adding overhead – we’re dedicated to focus on this.

## 5.15 Examples

The **examples** directory in the Circus repository contains a few examples to get you started.

Open a shell and cd into it:

```
$ cd examples
```

Now try to run the `example1.ini` config:

```
$ circusd example1.ini
2012-03-19 13:29:48 [7843] [INFO] Starting master on pid 7843
2012-03-19 13:29:48 [7843] [INFO] running dummy process [pid 7844]
2012-03-19 13:29:48 [7843] [INFO] running dummy process [pid 7845]
2012-03-19 13:29:48 [7843] [INFO] running dummy process [pid 7846]
2012-03-19 13:29:48 [7843] [INFO] running dummy process [pid 7847]
2012-03-19 13:29:48 [7843] [INFO] running dummy process [pid 7848]
2012-03-19 13:29:48 [7843] [INFO] running dummy2 process [pid 7849]
2012-03-19 13:29:48 [7843] [INFO] running dummy2 process [pid 7850]
2012-03-19 13:29:48 [7843] [INFO] running dummy2 process [pid 7851]
```

Congrats, you have 8 workers running !

Now run in a separate shell the listener script:

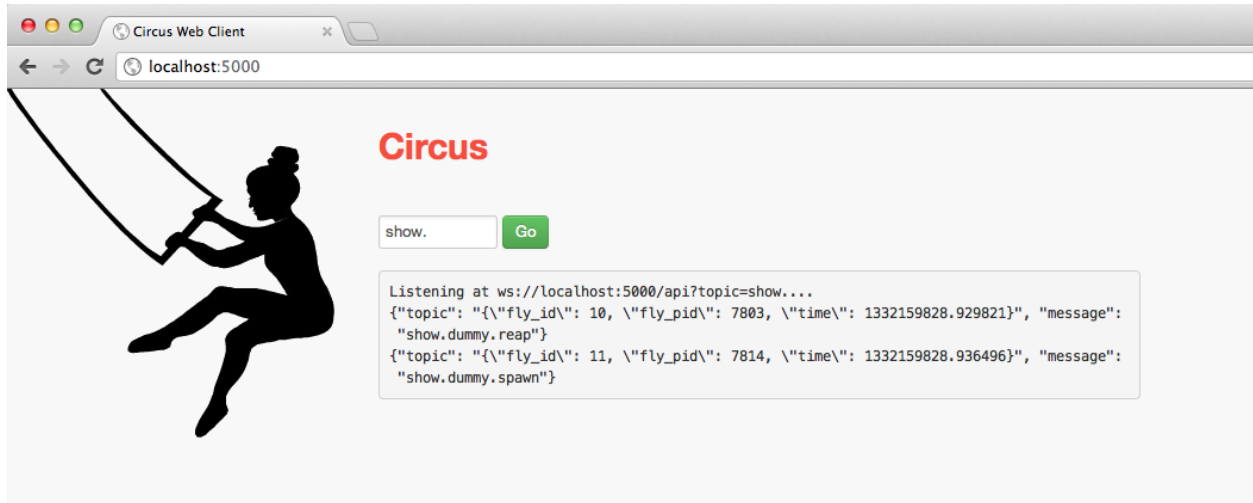
```
$ python listener.py
```

This script will print out all events happening in Circus. Try for instance to kill a worker:

```
$ kill 7849
```

You should see a few lines popping into the listener shell.

If you are brave enough, you can try the web socket demo with a web socket compatible browser. It will stream every event into the web page.



## 5.16 Circus Use Cases

This chapter presents a few use cases, to give you an idea on how to use Circus in your environment.

### 5.16.1 Running a WSGI application

Running a WSGI application with Circus is quite interesting because you can watch & manage your *web workers* using *circus-top*, *circusctl* or the Web interface.

This is made possible by using Circus sockets. See *Circus stack v.s. Classical stack*.

Let's take an example with a minimal Pyramid application:

```
from pyramid.config import Configurator
from pyramid.response import Response

def hello_world(request):
    return Response('Hello %(name)s!' % request.matchdict)

config = Configurator()
config.add_route('hello', '/hello/{name}')
config.add_view(hello_world, route_name='hello')
application = config.make_wsgi_app()
```

Save this script into an **app.py** file, then install those projects:

```
$ pip install Pyramid
$ pip install chaussette
```

Next, make sure you can run your Pyramid application using the **chaussette** console script:

```
$ chaussette app.application
Application is <pyramid.router.Router object at 0x10a4d4bd0>
Serving on localhost:8080
Using <class 'chaussette.backend._waitress.Server'> as a backend
```

And check that you can reach it by visiting **http://localhost:8080/hello/tarek**

Now that your application is up and running, let's create a Circus configuration file:

```
[circus]
check_delay = 5
endpoint = tcp://127.0.0.1:5555
pubsub_endpoint = tcp://127.0.0.1:5556
stats_endpoint = tcp://127.0.0.1:5557

[watcher:webworker]
cmd = chaussette --fd $(circus.sockets.webapp) app.application
use_sockets = True
numprocesses = 3

[socket:webapp]
host = 127.0.0.1
port = 8080
```

This file tells Circus to bind a socket on port *8080* and run *chaussette* workers on that socket – by passing its fd.

Save it to *server.ini* and try to run it using **circusd**

```
$ circusd server.ini
[INFO] Starting master on pid 8971
[INFO] sockets started
[INFO] circusd-stats started
[INFO] webapp started
[INFO] Arbiter now waiting for commands
```

Make sure you still get the app on **http://localhost:8080/hello/tarek**.

Congrats ! you have a WSGI application running 3 workers.

You can run the *The Web Console* or the *Command-line tools*, and enjoy Circus management.

## 5.16.2 Running a Django application

Running a Django application is done exactly like running a WSGI application.

Circus simply uses Chaussette’s ability to run Django applications. Instead of providing a fully qualified name for a WSGI python, you provide the path to the Django application, by prefixing it with **django::**

```
$ chaussette django:path/to/mysite
Application is <pyramid.router.Router object at 0x10a4d4bd0>
Serving on localhost:8080
Using <class 'chaussette.backend._waitress.Server'> as a backend
```

Chaussette will look for your settings file automatically for you, or you can explicitly give it with the **–django-settings** option:

```
$ chaussette django:path/to/mysite --backend gevent --django-settings mysite.settings
Application is <django.core.handlers.wsgi.WSGIHandler object at 0x10ec3f350>
Serving on localhost:8080
Using <class 'chaussette.backend._gevent.Server'> as a backend
```

See <http://chaussette.readthedocs.org> for more info on this.

## 5.17 Code coverage

Name	Stmts	Miss	Cover	Missing
-----				
/Users/tarek/Dev/github.com/circus/bin/bottle				1613 875 46% 25-36, 95-96
circus/__init__	41	30	27%	1-36, 106-118, 124
circus/arbiter	189	36	81%	95-106, 116, 137-138, 164, 167-179, 211-212, 221
circus/client	55	8	85%	17, 21, 49-50, 54-55, 60, 76
circus/commands/addwatcher	24	14	42%	1-66, 73, 78
circus/commands/base	72	55	24%	1-11, 19, 26, 35-79, 82, 86-97, 103-106
circus/commands/decrproc	17	14	18%	1-55, 60-63
circus/commands/dstats	24	23	4%	1-63, 66-81
circus/commands/get	25	19	24%	1-66, 76, 80-86
circus/commands/globaloptions	29	21	28%	1-73, 79-81, 93-99
circus/commands/incrproc	21	16	24%	1-53, 61-68
circus/commands/list	26	18	31%	1-54, 66-72
circus/commands/listsockets	13	10	23%	1-34, 40-45
circus/commands/numprocesses	19	17	11%	1-57, 59-60, 67-70
circus/commands/numwatchers	14	13	7%	1-42, 45-48
circus/commands/options	20	18	10%	1-101, 105-111
circus/commands/quit	7	6	14%	1-36
circus/commands/reload	17	15	12%	1-68, 70-71
circus/commands/restart	15	13	13%	1-56, 58-59
circus/commands/rmwatcher	12	10	17%	1-54
circus/commands/sendsignal	49	33	33%	1-123, 133, 135, 141-143, 148, 150, 156-163
circus/commands/set	34	22	35%	1-59, 70, 75
circus/commands/start	15	12	20%	1-53, 58
circus/commands/stats	49	41	16%	1-89, 93-99, 109-135
circus/commands/status	23	20	13%	1-65, 70-80
circus/commands/stop	12	8	33%	1-50
circus/commands/util	58	45	22%	1-38, 43, 47, 52, 55-56, 59-60, 64, 71-73
circus/config	130	49	62%	40-43, 70-71, 88-91, 107-119, 130-132, 135, 140
circus/consumer	32	10	69%	24, 28-31, 35, 38-42
circus/controller	116	13	89%	75, 85-86, 94-96, 104, 118, 142, 145, 151, 156
circus/plugins/__init__	140	101	28%	34-43, 47-55, 59-81, 85-93, 105-108, 118-119, 121
circus/process	145	42	71%	3-9, 103-123, 158-163, 181, 199-200, 223-224, 226
circus/py3compat	47	44	6%	1-38, 43-67
circus/sighandler	36	16	56%	34-44, 47, 50, 53, 56, 59
circus/sockets	54	13	76%	25, 42, 54-61, 70-71, 81
circus/stats/__init__	40	27	33%	34-81, 85
circus/stats/client	153	119	22%	28-33, 46-146, 151-157, 160, 163-166, 170-209, 211
circus/stats/collector	111	43	61%	9-10, 32, 37, 42-62, 65, 69-71, 73-89, 115-116, 118
circus/stats/publisher	25	18	28%	9-14, 17-28, 31-33
circus/stats/streamer	141	115	18%	20-38, 41, 44, 47-51, 55-56, 60-69, 72-97, 100-101
circus/stream/__init__	50	14	72%	16, 29, 34, 37-38, 41, 50, 53-59, 91
circus/stream/base	64	14	78%	22, 39, 51, 58-59, 65, 69-77
circus/stream/sthread	19	1	95%	25
circus/util	249	107	57%	1-57, 61-79, 85-87, 93, 107, 114, 123, 126, 153
circus/watcher	327	81	75%	142, 170, 180, 204, 230, 234, 244-245, 259-260, 262
circus/web/__init__	0	0	100%	
circus/web/circushttdp	166	89	46%	14-15, 52, 62, 72, 80, 90, 98, 104, 113, 118, 121
circus/web/controller	109	59	46%	9-11, 30, 48, 51, 55-56, 59-62, 65-66, 72-73, 75
base_html	NoSource: No source for code: '/Users/tarek/Dev/github.com/circus/docs/base_html': [Errno 2] No such file or directory			
connect_html	NoSource: No source for code: '/Users/tarek/Dev/github.com/circus/docs/connect_html': [Errno 2] No such file or directory			
index_html	NoSource: No source for code: '/Users/tarek/Dev/github.com/circus/docs/index_html': [Errno 2] No such file or directory			
-----				
TOTAL	4647	2387	49%	

## 5.18 Glossary

**arbiter** The *arbiter* is responsible for managing all the watchers within circus, ensuring all processes run correctly.

**controller** A *controller* contains the set of actions that can be performed on the arbiter.

**flapping** The *flapping detection* subscribes to events and detects when some processes are constantly restarting.

**process, worker, workers, processes** A *process* is an independent OS process instance of your program. A single watcher can run one or more processes. We also call them workers.

**pub/sub** Circus has a *pubsub* that receives events from the watchers and dispatches them to all subscribers.

**remote controller** The *remote controller* allows you to communicate with the controller via ZMQ to control Circus.

**watchers, watcher** A *watcher* is the program you tell Circus to run. A single Circus instance can run one or more watchers.

## 5.19 Contributing to Circus

Circus has been started at Mozilla but its goal is not to stay only there. We're trying to build a tool that's useful for others, and easily extensible.

We really are open to any contributions, in the form of code, documentation, discussions, feature proposal etc.

You can start a topic in our mailing list : <http://tech.groups.yahoo.com/group/circus-dev/>

Or add an issue in our [bug tracker](#)

### 5.19.1 Fixing typos and enhancing the documentation

It's totally possible that your eyes are bleeding while reading this half-english half-french documentation, don't hesitate to contribute any rephrasing / enhancement on the form in the documentation. You probably don't even need to understand how Circus works under the hood to do that.

### 5.19.2 Adding new features

New features are of course very much appreciated. If you have the need and the time to work on new features, adding them to Circus shouldn't be that complicated. We tried very hard to have a clean and understandable API, hope it serves the purpose.

You will need to add documentation and tests alongside with the code of the new feature. Otherwise we'll not be able to accept the patch.

### 5.19.3 How to submit your changes

We're using git as a DVCS. The best way to propose changes is to create a branch on your side (via *git checkout -b branchname*) and commit your changes there. Once you have something ready for prime-time, issue a pull request against this branch.

We are following this model to allow to have low coupling between the features you are proposing. For instance, we can accept one pull request while still being in discussion for another one.

Before proposing your changes, double check that they are not breaking anything! You can use the *tox* command to ensure this, it will run the testsuite under the different supported python versions.

### 5.19.4 Discussing

If you find yourself in need of any help while looking at the code of Circus, you can go and find us on irc at #mozilla-circus on irc.freenode.org (or if you don't have any IRC client, use [the webchat](#))

You can also start a thread in our mailing list - <http://tech.groups.yahoo.com/group/circus-dev>

## 5.20 How to add new commands in circus

If you want to add a new command, we tried to make this as simple as possible. You need to do three main things:

1. create a "your\_command.py" file under *circus/commands/*.
2. Implement a single class in there, with predefined methods
3. Add the new command in *circus/commands/\_\_init\_\_.py*.

Let's say we want to add a command which returns the number of watchers actually in use, we would do something like this (extensively commented to allow you to follow more easily):

```
class NumWatchers(Command):
    """It is a good practice to describe what the class does here.

    Have a look at other commands to see how we are used to format this
    text. It will be used to automatically appear in the documentation of
    circus, so don't be affraid of being exhaustive, that's what it is made
    for.
    """
    # all the commands need to inherit from `circus.commands.base.Command`

    name = "numwatchers"
    # you need to specify a name so we find back the command somehow

    options = [('', 'optname', default_value, 'description')]
    # XXX describe options

    properties = ['foo', 'bar']
    # properties list the command arguments that are mendatory. If they are
    # not provided, then an error will be thrown

    def execute(self, arbiter, props):
        # the execute method is the core of the command: put here all the
        # logic of the command and return a dict containing the values you
        # want to return, if any
        return {"numwatchers": arbiter.numwatchers()}

    def console_msg(self, msg):
        # msg is what is returned by the execute method.
        # this method is used to format the response for a console (it is
        # used for instance by circusctl to print its messages)
        return "a string that will be displayed"

    def validate(self, props):
        # this method is used to validate that the arguments passed to the
        # command are correct. An ArgumentError should be thrown in case
        # there is an error in the passed arguments (for instance if they
        # do not match together.
        # In case there is a problem wrt their content, a MessageError
```

```
# should be thrown. This method can modify the content of the props
# dict, it will be passed to execute afterwards.
```

## 5.21 Copyright

Circus was initiated by Tarek Ziade and is licenced under APLv2

Benoit Chesneau was an early contributor and did many things, like most of the circus.commands work.

### 5.21.1 Licence

Copyright 2012 - Mozilla Foundation  
Copyright 2012 - Benoit Chesneau

Licensed under the Apache License, Version 2.0 (the "License");  
you may not use this file except in compliance with the License.  
You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software  
distributed under the License is distributed on an "AS IS" BASIS,  
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.  
See the License for the specific language governing permissions and  
limitations under the License.

### 5.21.2 Contributors

In order of appearence:

- Tarek Ziadé
- Benoit Chesneau
- Pratyk S. Paudel
- Neil Chintomby
- Ori Livneh
- Pete Fritchman
- Johan Charpentier
- John Morrison
- Chris McDonald
- Stefane Fermigier
- Adnane Belmadi
- Alexis Métaireau
- Christian S. Perone





# CONTRIBUTIONS AND FEEDBACK

More on contribution: *Contributing to Circus*.

Useful Links:

- There's a mailing list for any feedback or question: <http://tech.groups.yahoo.com/group/circus-dev/>
- The repository and issue tracker is at GitHub : <https://github.com/mozilla-services/circus>
- Join us on the IRC : Freenode, channel **#mozilla-circus**