Circus Documentation

Release 0.4

Mozilla Foundation

June 25, 2012

CONTENTS

Circus is a process watcher and runner. It can be driven via a command-line interface or programmatically trough its python API.

It shares some of the goals of Supervisord, BluePill and Daemontools.

Circus is designed using Zero MQ. See *Design* for more details.

Note: Before running Circus, make sure you read the *Security* page.

To install it, check out *Requirements*

CHAPTER

ONE

USING CIRCUS VIA THE COMMAND-LINE

Circus provides a command-line script 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
```

Circus also provides two 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*

CHAPTER

TWO

MONITORING AND MANAGING CIRCUS THROUGH THE WEB

Circus provides 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 running:

```
$ circushttpd
```

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

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

CHAPTER

THREE

USING CIRCUS AS A LIBRARY

Circus provides high-level classes and functions that will let you manage processes. For example, if you want to run four workers forever, you can 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*

CHAPTER

FOUR

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*.

CHAPTER

FIVE

WHY SHOULD I USE CIRCUS INSTEAD OF X ?

1. **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 Supervisord 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*.

2. **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…

3. **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.

CHAPTER

SIX

MORE DOCUMENTATION

6.1 Requirements

- Python 2.6, 2.7 (3.x need to be tested)
- zeromq 2.10 or sup
- pyzmq >= 2.2.0

6.2 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*.

6.3 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


[watcher:myprogram]
cmd = python
args = -u myprogram.py $WID
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
```

6.3.1 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)

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)

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.

6.3.2 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

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)

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 instanciated, 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 instanciated, 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. Default to False.

6.3.3 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.

6.4 Command-line tools

6.4.1 circus-top

circus-top is a top-like console you can run to watch live your running Circus system. It will display the CPU and Memory usage.

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*.

6.4.2 circusctl

circusctl can be used to run any command listed below. For example, you can get a list of all the watchers:

```
$ circusctl list
```

circusctl is just a zeromq client, and if needed you can drive programmaticaly 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.

circus-ctl commands

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

- **reload**: *Reload the arbiter or a watcher*
- **numprocesses**: *Get the number of processes*
- **options**: *Get the value of a watcher option*

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

ZMQ Message

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

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

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

Command line

```
$ circusctl descr <name>
```

Options

- <name>: name of the watcher

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 Optios 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": "val1",
        ..
    }
}
```

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>"
    }
}
```

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

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

Command line

```
$ circusctl incr <name>
```

Options

- <name>: name of the watcher

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 processes in a watcher:

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

The response return the list asked. Flies returned are process ID that can be used in others commands.

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 suscriber 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 list of pids in a watcher

ZMQ Message To get the list of pid in a watcher:

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

The response return the list asked.

Command line

```
$ circusctl listpids <name>
```

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": "val1",
        ..
    }
}
```

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 the signal to all processes in a watcher, a specific process in a watcher or its children

ZMQ Message To get the list of all process in the watcher:

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

To send a signal to a process:

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

An optionnal property "children" can be used to send the signal to all the children rather than the process itself.

To send a signal to a process child:

To get the list of processes in a watcher:

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

The response return the status "ok".

Command line

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

Options:

- <name>: the name of the watcher
- <process>: the process id. You can get them using the command list
- <pid>: integer, the process id.
- <signum> : the signal number to send.

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 start 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, 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 teh 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 retun an object per process with 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"
  },
  "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",
    "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 stop [<name>]
```

Options

- <name>: name of the watcher

6.5 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 enpoint 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:

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

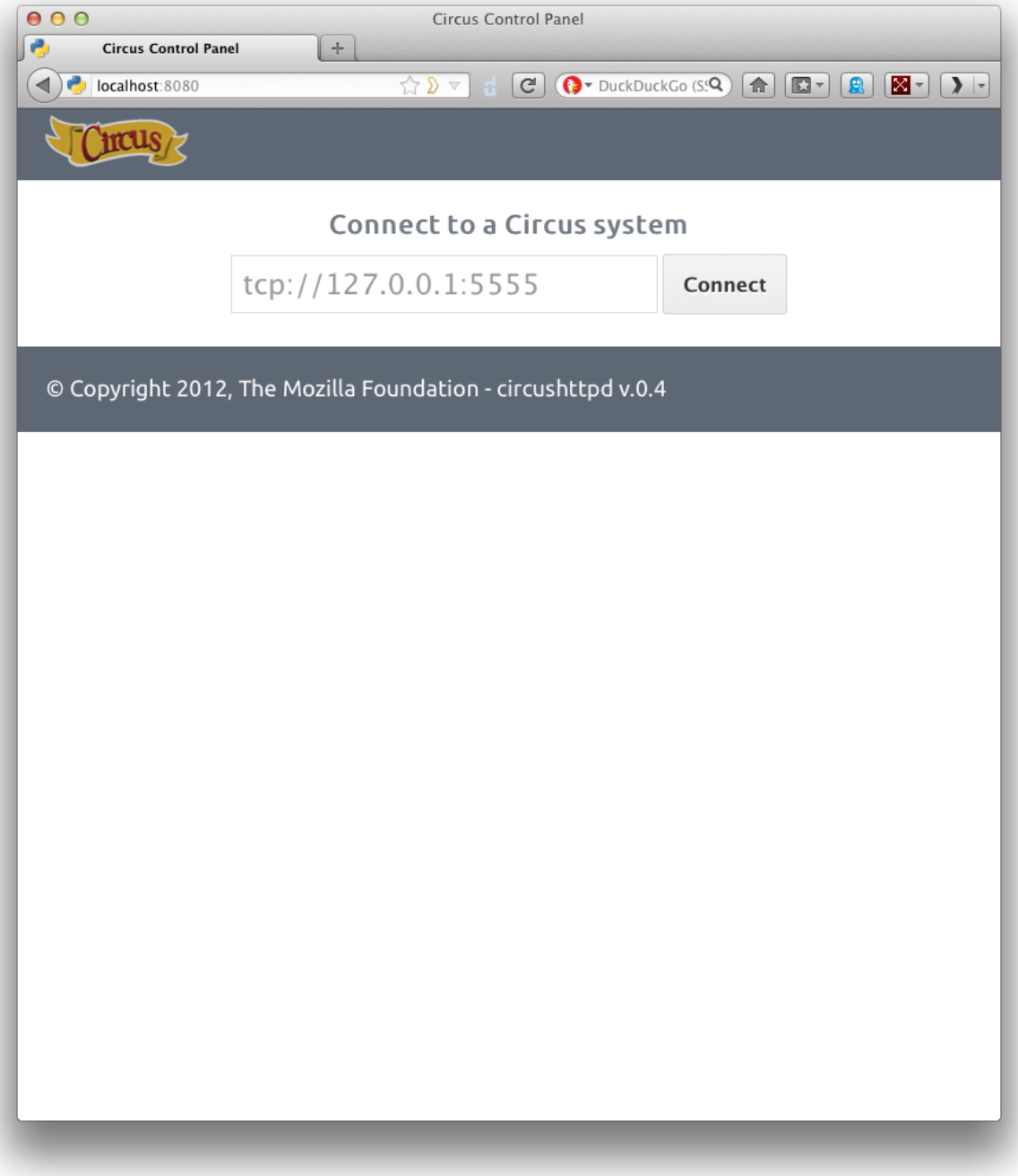
To enable the console, run the **circushttpd** script:

```
$ circushttpd
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.

6.5.1 Using the console

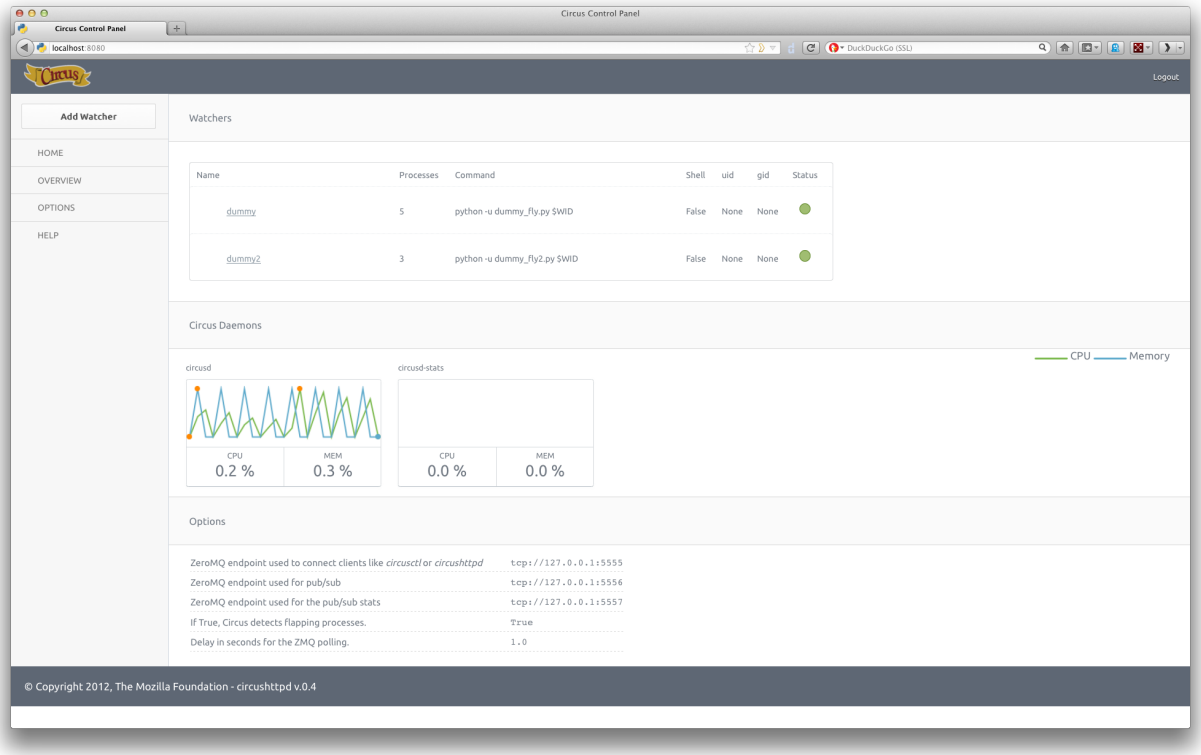
Once the script is running, you can open a browser and visit *http://localhost:8080*. You should get this screen:



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*.

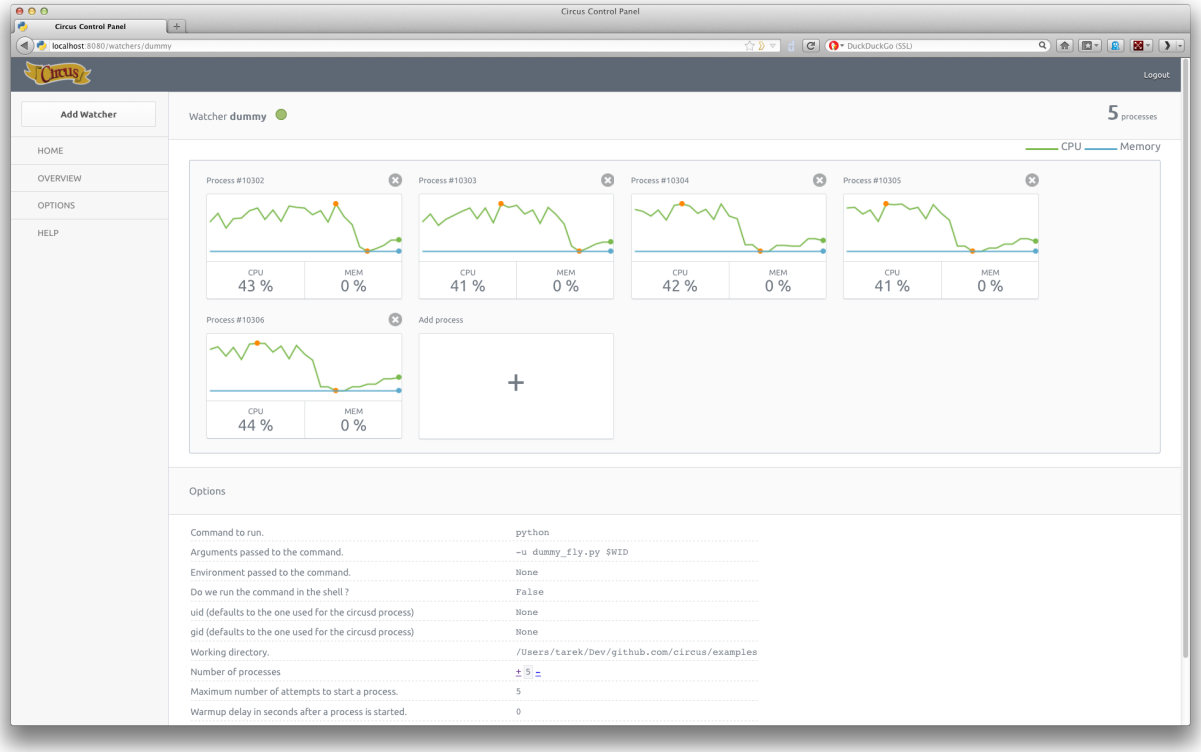
Once you hit *Connect*, the web application will connect to the Circus system.

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).



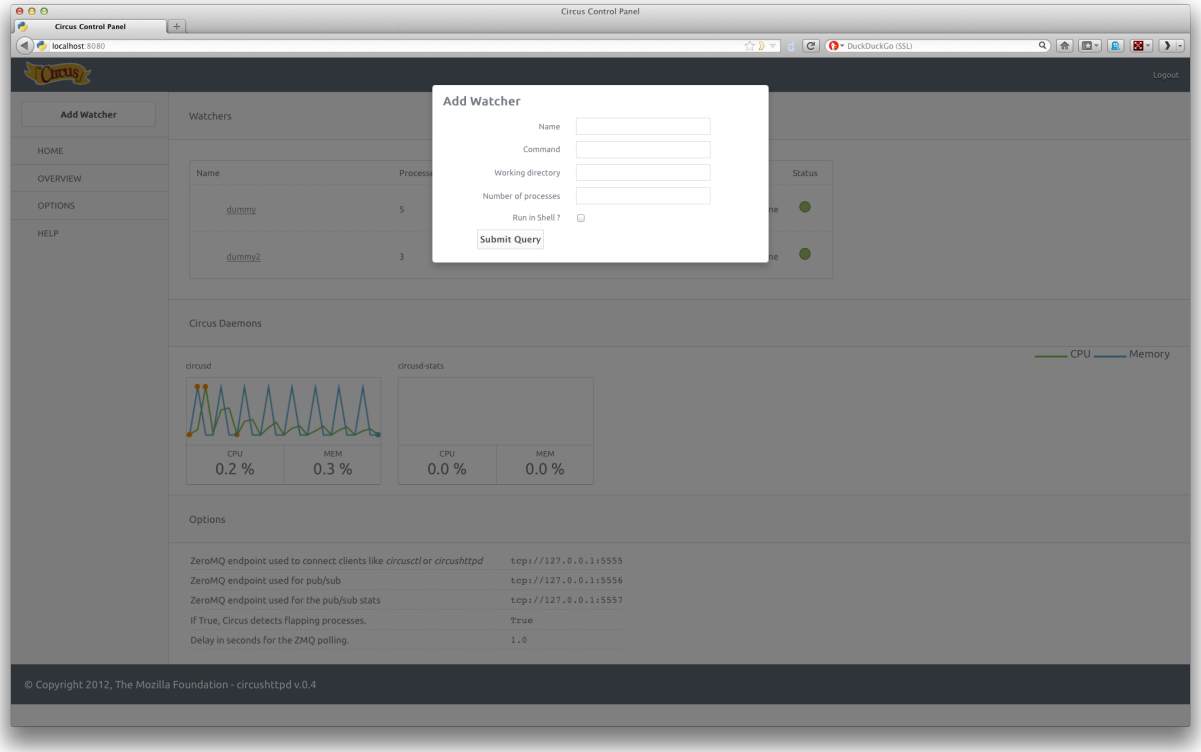
You can click on the status of each watcher to toggle it from **Active** (green) to **Inactive** (red). This change is effective immediatly and let you start & stop watchers.

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:



6.5.2 Running behind Nginx & Gunicorn

circushttpd 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.

A nice combo is Gunicorn & Nginx:

- Gunicorn is the WSGI web server and serves the Web application on the 8080 port.
- Nginx acts as a proxy in front of Gunicorn. It an also deal with security.

Gunicorn

To run Gunicorn, make sure Gunicorn is installed in your environment and simply use the **–server** option:

```
$ pip install gunicorn
$ bin/circushttpd --server gunicorn
Bottle server starting up (using GunicornServer())...
Listening on http://localhost:8080/
Hit Ctrl-C to quit.

2012-05-14 15:10:54 [13536] [INFO] Starting gunicorn 0.14.2
2012-05-14 15:10:54 [13536] [INFO] Listening at: http://127.0.0.1:8080 (13536)
2012-05-14 15:10:54 [13536] [INFO] Using worker: sync
2012-05-14 15:10:54 [13537] [INFO] Booting worker with pid: 13537
```

If you want to use another server, you can pick any server listed in http://bottlepy.org/docs/dev/tutorial.html#multi-threaded-server

Nginx

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

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;
}
```

If you want a more complete Nginx configuration example, have a look at : http://gunicorn.org/deploy.html

6.5.3 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.

6.5.4 Extending the web console

We chosed to use 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 doa 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.

You may also encounter the StatsNamespace class. It's the class which manages the websocket communication on the server side. Its documentation should help you to understand what it does.

6.6 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.

6.6.1 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'*, *pubsub_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()
```

6.6.2 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*)

Wraps a process.

Options:

- **wid**: the process unique identifier. This value will be used to replace the *$WID* string in the command line if present.
- **cmd**: the command to run. May contain *$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.
- **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.

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
- 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*, ***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 callable that will receive the stream of the process stdout. Defaults to None.

  When provided, *stdout_stream* is a mapping containing two keys:

  –stream: the callable that will receive the updates streaming. Defaults to `circus.stream.FileStream`

  –refresh_time: the delay between two stream checks. Defaults to 0.3 seconds.

  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 callable that will receive the stream of the process stderr. Defaults to None.

  When provided, *stdout_stream* is a mapping containing two keys:

  –stream: the callable that will receive the updates streaming. Defaults to `circus.stream.FileStream`

  –refresh_time: the delay between two stream checks. Defaults to 0.3 seconds.

  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)
- **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.

send_msg(*topic*, *msg*)
send msg

reap_processes(**args*, ***kw*)
Reap processes.

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 processes.

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*, *prereload_fn=None*, *context=None*, *loop=None*, *stats_endpoint=None*, *plugins=None*)
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

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 gracefuly 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.

6.7 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 emited by circusd
- the flapping feature which avoid to re-launch processes infinitely when they die too quickly.
- many more to come !

6.7.1 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 befor 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.

6.7.2 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.

6.7.3 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.Logge
[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*.

6.7.4 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.

6.8 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.

6.9 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 recommandations if you want to protect your server.

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

6.9.1 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.

6.9.2 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.

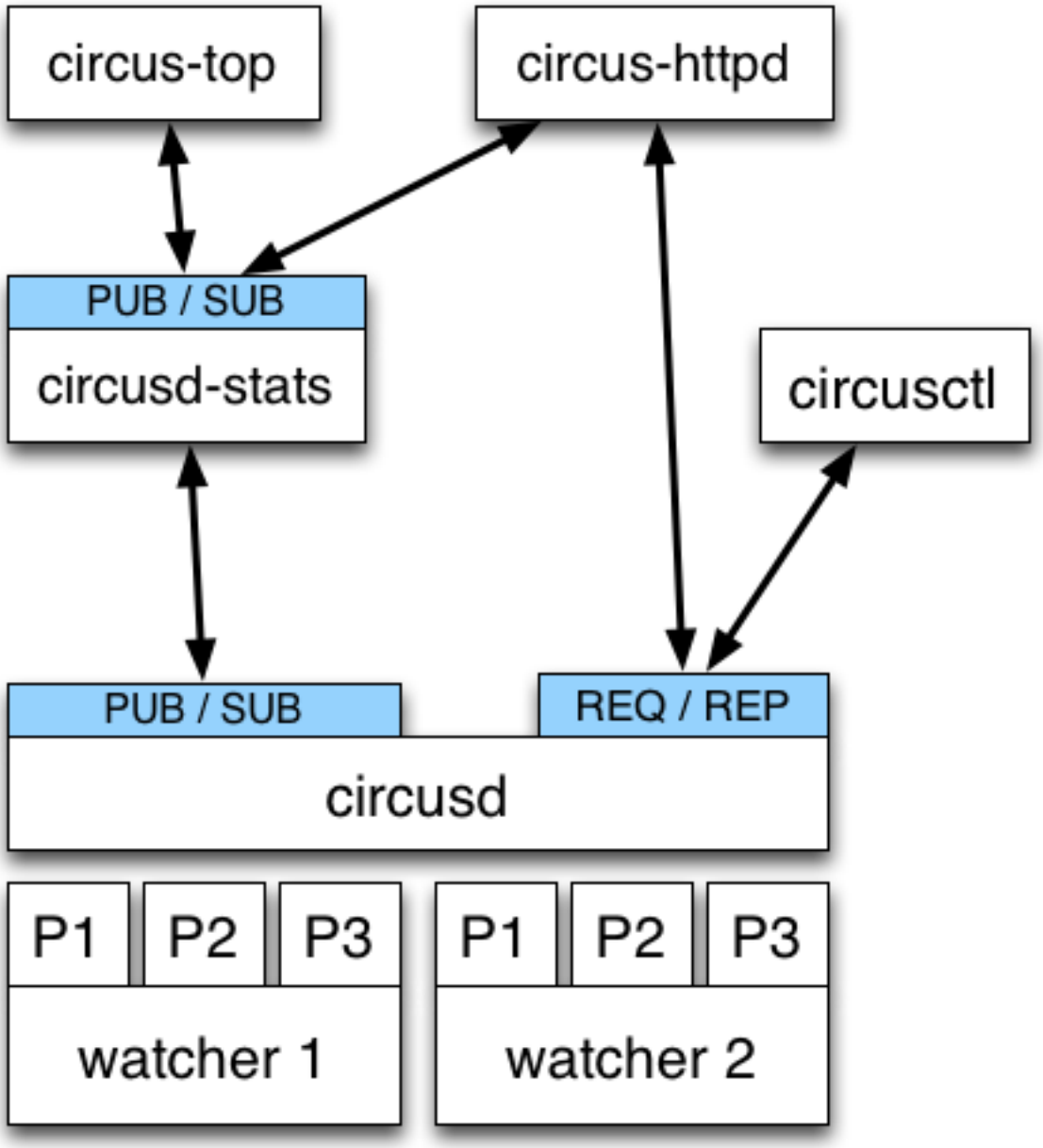
6.9.3 circushttpd

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.

6.10 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 managment 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.

6.11 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:

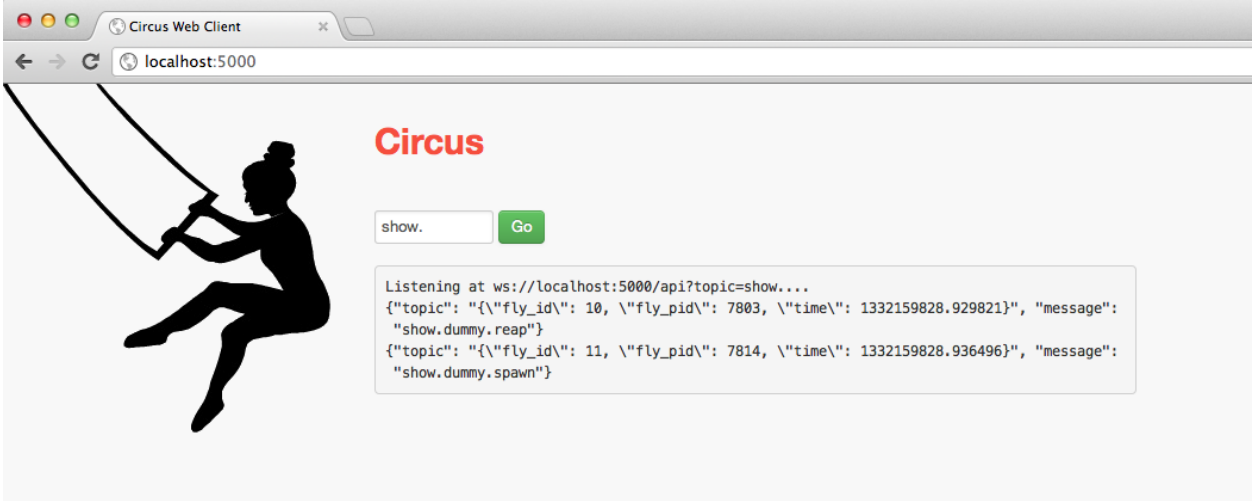
```
$ python listener.py
```

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.



6.12 Code coverage

```
Name                                 Stmts   Miss  Cover   Missing
------------------------------------------------------------------
/Users/tarek/Dev/github.com/circus-master/bin/bottle                      1613   1126    30%   25-36
circus/__init__                         40     29    28%   1-35, 105-117, 123
circus/arbiter                         176     29    84%   66-72, 86-102, 150-154, 187-188, 193, 213, 217-
circus/client                           55      9    84%   18, 22, 50-51, 55-56, 65, 76-77
circus/commands/addwatcher              24     14    42%   1-66, 73, 78
circus/commands/base                    72     53    26%   1-11, 19, 26, 35-61, 65-79, 82, 86-97, 103-106
circus/commands/decrproc                16     14    13%   1-53, 57-60
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                20     16    20%   1-51, 58-65
circus/commands/list                    23     17    26%   1-52, 61-67
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              47     33    30%   1-109, 114, 118, 124, 127, 130, 138-147
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                    54     42    22%   1-38, 43, 47, 52, 55-56, 59-60, 64
circus/config                          129     59    54%   40-43, 55, 58-61, 72-96, 101-104, 119-131, 141,
circus/controller                      116     16    86%   75, 85-86, 94-96, 104, 116-119, 122, 142, 145,
circus/plugins/__init__                140    101    28%   34-43, 47-55, 59-81, 85-93, 105-108, 118-119, 1
circus/process                         122     40    67%   3-9, 92, 97, 100-120, 147, 164-165, 188-189, 19
circus/py3compat                        47     44     6%   1-38, 43-67
circus/sighandler                       36     10    72%   39-44, 47, 50, 53, 59
circus/stats/__init__                   37     27    27%   23-71, 75
circus/stats/collector                  98     13    87%   30, 34-35, 40, 56-60, 127-131, 138-139
circus/stats/publisher                  43      5    88%   37-40, 43-44
```

```
circus/stats/streamer              117     41    65%   47, 68, 72-75, 78-82, 96-110, 113-135
circus/stream/__init__              35      7    80%   16, 29, 34, 37-38, 41, 68
circus/stream/base                  64     11    83%   22, 39, 51, 58-59, 64-65, 74-77
circus/stream/sthread               19      1    95%   25
circus/util                        215     96    55%   1-56, 60-78, 84-86, 92, 106-113, 120, 160-161,
circus/watcher                     318     67    79%   132, 159, 169, 225, 251, 255, 276, 280, 283-284
circus/web/__init__                  0      0   100%
circus/web/circushttpd             131    121     8%   11-12, 19-281
------------------------------------------------------------
TOTAL                             4098   2276    44%
```

6.13 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.

6.14 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 : https://lists.mozilla.org/listinfo/dev-services-circus

Or add an issue in our bug tracker

6.14.1 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.

6.14.2 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.

6.14.3 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.

6.14.4 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: https://lists.mozilla.org/listinfo/dev-services-circus

6.15 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.

6.15.1 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.
```

6.15.2 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 Belmadiaf
- Alexis Métaireau
- Christian S. Perone

CHAPTER

SEVEN

CONTRIBUTIONS AND FEEDBACK

More on contribution: *Contributing to Circus*.

Useful Links:

- There's a maling list for any feedback or question: https://lists.mozilla.org/listinfo/dev-services-circus
- The repository and issue tracker is at GitHub : https://github.com/mozilla-services/circus
- Join us on the IRC : Freenode, channel **#mozilla-circus**