pgctl Documentation

Release 3.0.1

Buck Evan

Oct 25, 2017

Contents

1 Introduction **3**

2 Feature Support **5**

3 User Guide **7**
- 3.1 Installation . . . 7
- 3.2 Quickstart . . . 8
- 3.3 Sub-Commands . . . 9
- 3.4 Advanced Usage . . . 10

4 API Documentation **13**
- 4.1 API Documentation . . . 13

5 Contributor Guide **17**
- 5.1 Contributor's guide . . . 17
- 5.2 Design Rationale . . . 19
- 5.3 Bug Log . . . 20

Python Module Index **21**

Issues | Github | PyPI

Release v3.0. (*Installation*)

CHAPTER 1

Introduction

`pgctl` is an MIT Licensed tool to manage developer "playgrounds".

Often projects have various processes that should run in the backround (*services*) during development. These services amount to a miniature staging environment that we term *playground*. Each service must have a well-defined state at all times (it should be starting, up, stopping, or down), and should be independantly restartable and debuggable.

`pgctl` aims to solve this problem in a unified, language-agnostic framework (although the tool happens to be written in Python).

As a simple example, let's say that we want a *date* service in our playground, that ensures our *now.date* file always has the current date.

```
$ cat playground/date/run
date > now.date

$ pgctl start
$ pgctl status
date -- up (0 seconds)

$ cat now.date
Fri Jun 26 15:21:26 PDT 2015

$ pgctl stop
$ pgctl status
date -- down (0 seconds)
```

CHAPTER 2

Feature Support

- User-friendly Command Line Interface
- Simple Configuration
- Python 2.7—3.5

CHAPTER 3

User Guide

This part of the documentation covers the step-by-step instructions and usage of `pgctl` for getting started quickly.

Installation

This part of the documentation covers the installation of pgctl. The first step to using any software package is getting it properly installed.

Distribute & Pip

Installing pgctl is simple with pip, just run this in your terminal:

```
$ pip install pgctl
```

Get the Code

pgctl is actively developed on GitHub, where the code is always available.

You can either clone the public repository:

```
$ git clone git://github.com/yelp/pgctl.git
```

Download the tarball:

```
$ curl -OL https://github.com/yelp/pgctl/tarball/master
```

Or, download the zipball:

```
$ curl -OL https://github.com/yelp/pgctl/zipball/master
```

Once you have a copy of the source, you can embed it in your Python package, or install it into your site-packages easily:

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

Quickstart

This page attempts to be a quick-and-dirty guide to getting started with pgctl.

Setting up

The minimal setup for pgctl is a `playground` directory containing the services you want to run. A service consists of a directory with a `run` script. The script should run in the foreground.

```
$ cat playground/date/run
date > now.date
```

Once this is in place, you can start your playground and see it run.

```
$ pgctl start
$ pgctl log
[webapp] Serving HTTP on 0.0.0.0 port 36474 ...

$ curl
```

Writing Playground Services

`pgctl` works best with a single process. When writing a `run` script in bash, use the `exec` statement to replace the shell with your process. This avoids a process tree with bash as the parent of your service. Having a single process allows simple management of state and proper signalling for stopping the service.

Bad: (don't do this!)

```
#!/bin/bash
sleep infinity  # creates a new process
```

Good: (do it this way!)

```
#!/bin/bash
exec sleep infinity  # replaces the *current* process
```

Without the exec, stopping the service will kill *bash* but the *sleep* process will be left behind. This kind of process-tree management is too complex for `pgctl` to auto-magically fix it for you, but it will let you know if it becomes a problem:

```
$ pgctl restart
Stopping: sleeper
Stopped: sleeper
ERROR: We sent SIGTERM, but these processes did not stop:
                     USER        PID ACCESS COMMAND
playground/sleeper:    buck     2847827 f.c.. sleep

To fix this temporarily, run: pgctl stop sleeper --force
```

```
To fix it permanently, see:
    http://pgctl.readthedocs.org/en/latest/user/quickstart.html#writing-playground-
↪services
```

Aliases

With no arguments, `pgctl start` is equivalent to `pgctl start default`. By default, `default` maps to a list of all services. You can configure what `default` means via `pgctl.yaml`:

```
aliases:
    default:
        - service1
        - service2
```

You can also add other aliases this way. When you name an alias, it simply expands to the list of configured services, so that `pgctl start A-and-B` would be entirely equivalent to `pgctl start A B`.

Sub-Commands

`pgctl` has eight basic commands: `start`, `stop`, `restart`, `debug`, `status`, `log`, `reload`, `config`

Note: With no arguments, `pgctl <cmd>` is equivalent to `pgctl <cmd> default`. By default, default maps to all services. See *Aliases*.

---

start

```
$ pgctl start <service=default>
```

Starts a specific service, group of services, or all services. This command is blocking until all services have successfully reached the up state. `start` is idempotent.

stop

```
$ pgctl stop <service=default>
```

Stops a specific service, group of services, or all services. This command is blocking until all services have successfully reached the down stated. `stop` is idempotent.

restart

```
$ pgctl restart <service=default>
```

Stops and starts specific service, group of services, or all services. This command is blocking until all services have successfully reached the down stated.

debug

```
$ pgctl debug <service=default>
```

Runs a specific service in the foreground.

status

```
$ pgctl status <service=default>
<service> (pid <PID>) -- up (0 seconds)
```

Retrieves the state, PID, and time in that state of a specific service, group of services, or all services.

log

```
$ pgctl log <service=default>
```

Retrieves the stdout and stderr for a specific service, group of services, or all services.

reload

```
$ pgctl reload <service=default>
```

Reloads the configuration for a specific service, group of services, or all services.

config

```
$ pgctl config <service=default>
```

Prints out a configuration for a specific service, group of services, or all services.

Advanced Usage

You may (or may not) want these notes after using pgctl for a while.

Services that stop slowly

When you have a service that takes a while to stop, pgctl may incorrectly error out saying that the service left processes behind. By default, pgctl only waits up to two seconds. To tell pgctl to wait a bit longer write a number of seconds into a `timeout-stop` file.

```
$ echo 10 > playground/uwsgi/timeout-stop
$ git add playground/uwsgi/timeout-stop
```

Services that start slowly

Similarly, if pgctl needs to be told to wait longer to start your service, write a `timeout-ready` file.

If there's a significant period between when the service has started (up) and when it's actually doing it's job (ready), or if your service sometimes stops working even when it's running, create a runnable `ready` script in the service directory and prefix your service command with our `pgctl-poll-ready` helper script. `pgctl-poll-ready` will run the `ready` script repeatedly to determine when your service is actually ready. As an example:

```
$ cat playground/uwsgi/run
make -C ../../ minimal  # the build takes a few seconds
exec pgctl-poll-ready ../../bin/start-dev

$ cat playground/uwsgi/ready
exec curl -s localhost:9003/status

$ cat playground/uwsgi/timeout-ready
30
```

Handling subprocesses in a bash service

If you're unable to use `exec` to *create a single-process service*, you'll need to handle `SIGTERM` and kill off your subprocesses yourself. In bash this is tricky. See the example in our test suite for an example of how to do this reliably:

https://github.com/Yelp/pgctl/blob/master/tests/examples/output/playground/ohhi/run

CHAPTER 4

API Documentation

If you are looking for information on a specific function, class or method, this part of the documentation is for you.

API Documentation

This is automatically generated documentation from the source code. Generally this will only be useful for developers.

Submodules

pgctl.cli module

class pgctl.cli.**PgctlApp**(*config=<frozendict {u'force': False, u'verbose': False, u'pgdir': u'playground', u'json': False, u'timeout': u'2.0', u'services': (u'default',), u'poll': u'.01', u'pghome': u'~/.run/pgctl', u'aliases': <frozendict {u'default': (u'(all services)',)}>}>*)

Bases: object

all_services
Return a list of all services.

Returns list of Service objects

Return type list

commands = (<function start>, <function stop>, <function status>, <function restart>, <function reload>, <function log>,

config()
Print the configuration for a service

debug()
Allow a service to run in the foreground

log(*interactive=None*)
Displays the stdout and stderr for a service or group of services

pgdir
Retrieve the set playground directory

pghome
Retrieve the set pgctl home directory.

By default, this is "$XDG_RUNTIME_DIR/pgctl".

playground_locked(**args*, ***kwds*)
Lock the entire playground.

reload()
Reloads the configuration for a service

restart()
Starts and stops a service

service_by_name(*service_name*)
Return an instantiated Service, by name.

service_names

services
Return a tuple of the services for a command

Returns tuple of Service objects

start()
Idempotent start of a service or group of services

status()
Retrieve the PID and state of a service or group of services

stop(*with_log_running=False*)
Idempotent stop of a service or group of services

Parameters **with_log_running** – controls whether the logger associated with

this service should be stopped or left running. For restart cases, we want to leave the logger running (since poll-ready may still be writing log messages).

with_services(*services*)
return a similar PgctlApp, but with a different set of services

class pgctl.cli.**Start**(*service*)
Bases: *pgctl.cli.StateChange*

assert_()

change()

fail()

get_timeout()

is_user_facing = True

class strings
Bases: object

change = u'start'

changed = u'Started:'

changing = u'Starting:'

class pgctl.cli.**StateChange**(*service*)
Bases: object

class pgctl.cli.**StateChangeResult**
Bases: object

FAILURE = 1

RECHECK_NEEDED = 2

SUCCESS = 0

class pgctl.cli.**Stop**(*service*)
Bases: *pgctl.cli.StateChange*

assert_()

change()

fail()

get_timeout()

is_user_facing = True

class strings
Bases: object

change = u'stop'

changed = u'Stopped:'

changing = u'Stopping:'

class pgctl.cli.**StopLogs**(*service*)
Bases: *pgctl.cli.StateChange*

assert_()

change()

fail()

get_timeout()

is_user_facing = False

class strings
Bases: object

change = u'stop'

changed = u'Stopped logger for:'

changing = u'Stopping logger for:'

class pgctl.cli.**TermStyle**
Bases: object

BOLD = u'\x1b[1m'

ENDC = u'\x1b[0m'

GREEN = u'\x1b[92m'

RED = u'\x1b[91m'

YELLOW = u'\x1b[93m'

classmethod **wrap**(*text*, *style*)

pgctl.cli.**error_message_on_timeout**(*service*, *error*, *action_name*, *actual_timeout_length*, *check_length*)

pgctl.cli.**main**(*argv=None*)

pgctl.cli.**parser**()

pgctl.cli.**pgctl_print**(**print_args*, ***print_kwargs*)
Print to stderr with [pgctl] prepended.

pgctl.cli.**timeout**(*service*, *start_time*, *check_time*, *curr_time*)

pgctl.cli.**unbuf_print**(**args*, ***kwargs*)
Print unbuffered in utf8.

Module contents

CHAPTER 5

Contributor Guide

If you want to contribute to the project, this part of the documentation is for you.

Contributor's guide

This page helps you make contributions to the pgctl project.

For a quick primer on using github, see https://guides.github.com/activities/contributing-to-open-source/

Developer Environment

To put yourself into our dev environment, run `source .activate.sh`.

Documentation

If you need to make changes to the documentation, they live under docs/source. For a quick primer on the rst format, see http://docutils.sourceforge.net/docs/user/rst/quickref.html

If you want to see good examples of other projects' documentation, see:

- [the Requests docs](https://github.com/kennethreitz/requests/tree/master/docs)
- [the virtualenv docs](https://github.com/pypa/virtualenv/tree/master/docs)

To get a look at your changes, run *make docs* from the root of the project. This will spin up a http server on port 8088 serving your editted documentation.

Debugging

To get extra debugging output from pgctl, set the `PGCTL_VERBOSE` environment variable. This will cause any tests that assert the *output* of `pgctl` to fail, but it often helps finding mysterious issues.

Run tests

- make test ## (should Just Work)
- tox -e test ## lose proper –recreate logic
- ./test ## *python* must have all test deps
- py.test ## lose coverage and linting

Filter which tests to run

- make test ARGS='-k "test and stop"'
- tox -e test – '-k "test and stop"'
- ./test -k "test and stop"
- py.test tests -k "test and stop"

Run a particular test

- py.test tests/main_test.py::test_stop

Coverage reports should show all project files as well as test files.

Looking at Coverage

It's good practice to look at unit coverage separately from spec coverage. First,

    make unit test

or:

    make spec test

And in a separate terminal:

    make coverage-server

Complications

These are the things that make things more complicated than they (seem to) need to be.

A broken setup.py should cause failing tests. Many projects' testing setup will blissfully pass even if setup.py does nothing whatsoever. In order to avoid this, I use *changedir* in my tox.ini. Most of the other complexity comes from this. For example, because I run the code that's inside the virtualenv during test, it's fiddly to get coverage to report on the right copy of the code.

Subprocess coverage is complicated. coveragepy has some built-in support for this, but it's not enabled by default. The script at *tests/testing/install_coverage_pth.py* does the necessary addtional work to enable the subprocess coverage feature. Because several coverage runs may be running concurrently, we must be careful to always use coverage in "parallel mode" and run *coverage combine* afterward.

Design Rationale

Directory Structure

```
$ pwd
/home/<user>/<project>

$ tree playground/
playground/
- service1
|   - down
|   - run
|   - stderr.log
|   - stdout.log
|   - supervise -> ~/.run/pgctl/home/<user>/<project>/playground/service1/supervise
- service2
|   - down
|   - run
|   - stderr.log
|   - stdout.log
|   - supervise -> ~/.run/pgctl/home/<user>/<project>/playground/service2/supervise
- service3
    - down
    - run
    - stderr.log
    - stdout.log
    - supervise -> ~/.run/pgctl/home/<user>/<project>/playground/service3/supervise
```

There are a few points to note: logging, services, state, symlinking.

logging

stdin and stdout will be captured from the supervised process and written to log files under the service directory. The user will be able to use the `pgctl log` command to aggregate these logs in a readable form.

services

All services are located under the playground directory.

state

We are using s6 for process management and call the `s6-supervise` command directly. It was a design decision to not use `svscan` to automatically supervise all services. This was due to inflexability with logging (by default stdout is only logged). To ensure that every service is in a consistent state, a down file is added to each service directory (man supervise) if it does not already exist.

symlinking

Currently `pip install .` calls shutil.copy to copy all files in the current project when in the project's base directory. Having pipes present in the projects main directory attempts to copy the pipe and deadlocks. To remedy this situation, we have symlinked the supervise directory to the user's home directory to prevent any pip issues.

–force option

`--force` takes effect only upon `pgctl stop`, not `pgctl start`. `--force` implies that pgctl would try whatever it can to accomplish a task. This would not apply to `pgctl start` under many cases. For example, if a service takes 30 minutes to warm itself up before ready, pgctl cannot force it to start up within a short period of time. Instead, users should take the responsibility to adjust the timeout value.

Design Decisions

Design of debug

Unsupervise all things when down

Bug Log

This documents current and past bugs in the project. This is helpful when during future debugging sessions.

Current bugs – 2015-10-26

Currently the coverage report improperly shows missing coverage, but only under jenkins / circleCI. Local testing and travis don't seem to have this issue.

I've found some clues:

- only lines run *directly* by the xdist workers goes missing; all subprocess coverage is reliable.
- the xdist worker does write out its coverage file on time, it's just (mostly) empty.
- **from looking at the coverage debugging trace: the coverage drops out at this line:** https://bitbucket.org/hpk42/execnet/src/50f88cb892d/execnet/gateway_base.py#gateway_base.py-1072
- TODO: does this reproduce using coverage<4.0 ?

Circle CI debugging

To grab files from a circleCI run: (for example)

> rsync -Pav -e 'ssh -p 64785' ubuntu@54.146.184.147:pgctl/coverage.bak.2015-10-24_18:28:36.937047774 .

Python Module Index

p

pgctl, 16
pgctl.cli, 13

Index

A

all_services (pgctl.cli.PgctlApp attribute), 13
assert_() (pgctl.cli.Start method), 14
assert_() (pgctl.cli.Stop method), 15
assert_() (pgctl.cli.StopLogs method), 15

B

BOLD (pgctl.cli.TermStyle attribute), 15

C

change (pgctl.cli.Start.strings attribute), 14
change (pgctl.cli.Stop.strings attribute), 15
change (pgctl.cli.StopLogs.strings attribute), 15
change() (pgctl.cli.Start method), 14
change() (pgctl.cli.Stop method), 15
change() (pgctl.cli.StopLogs method), 15
changed (pgctl.cli.Start.strings attribute), 14
changed (pgctl.cli.Stop.strings attribute), 15
changed (pgctl.cli.StopLogs.strings attribute), 15
changing (pgctl.cli.Start.strings attribute), 14
changing (pgctl.cli.Stop.strings attribute), 15
changing (pgctl.cli.StopLogs.strings attribute), 15
commands (pgctl.cli.PgctlApp attribute), 13
config() (pgctl.cli.PgctlApp method), 13

D

debug() (pgctl.cli.PgctlApp method), 13

E

ENDC (pgctl.cli.TermStyle attribute), 15
error_message_on_timeout() (in module pgctl.cli), 16

F

fail() (pgctl.cli.Start method), 14
fail() (pgctl.cli.Stop method), 15
fail() (pgctl.cli.StopLogs method), 15
FAILURE (pgctl.cli.StateChangeResult attribute), 15

G

get_timeout() (pgctl.cli.Start method), 14
get_timeout() (pgctl.cli.Stop method), 15
get_timeout() (pgctl.cli.StopLogs method), 15
GREEN (pgctl.cli.TermStyle attribute), 15

I

is_user_facing (pgctl.cli.Start attribute), 14
is_user_facing (pgctl.cli.Stop attribute), 15
is_user_facing (pgctl.cli.StopLogs attribute), 15

L

log() (pgctl.cli.PgctlApp method), 13

M

main() (in module pgctl.cli), 16

P

parser() (in module pgctl.cli), 16
pgctl (module), 16
pgctl.cli (module), 13
pgctl_print() (in module pgctl.cli), 16
PgctlApp (class in pgctl.cli), 13
pgdir (pgctl.cli.PgctlApp attribute), 13
pghome (pgctl.cli.PgctlApp attribute), 14
playground_locked() (pgctl.cli.PgctlApp method), 14

R

RECHECK_NEEDED (pgctl.cli.StateChangeResult attribute), 15
RED (pgctl.cli.TermStyle attribute), 15
reload() (pgctl.cli.PgctlApp method), 14
restart() (pgctl.cli.PgctlApp method), 14

S

service_by_name() (pgctl.cli.PgctlApp method), 14
service_names (pgctl.cli.PgctlApp attribute), 14
services (pgctl.cli.PgctlApp attribute), 14
Start (class in pgctl.cli), 14

start() (pgctl.cli.PgctlApp method), 14
Start.strings (class in pgctl.cli), 14
StateChange (class in pgctl.cli), 14
StateChangeResult (class in pgctl.cli), 15
status() (pgctl.cli.PgctlApp method), 14
Stop (class in pgctl.cli), 15
stop() (pgctl.cli.PgctlApp method), 14
Stop.strings (class in pgctl.cli), 15
StopLogs (class in pgctl.cli), 15
StopLogs.strings (class in pgctl.cli), 15
SUCCESS (pgctl.cli.StateChangeResult attribute), 15

T

TermStyle (class in pgctl.cli), 15
timeout() (in module pgctl.cli), 16

U

unbuf_print() (in module pgctl.cli), 16

W

with_services() (pgctl.cli.PgctlApp method), 14
wrap() (pgctl.cli.TermStyle class method), 15

Y

YELLOW (pgctl.cli.TermStyle attribute), 15