honcho Documentation

Release 1.0.1

Nick Stenning

Jun 27, 2017

Contents

Welcome! This is the home of Honcho and its documentation. Honcho is:

1. A Python port of David Dollar's Foreman: a command-line application which helps you manage and run Procfile-based applications. It helps you simplify deployment and configuration of your applications in both development and production environments.
2. Secondarily, Honcho is a Python library/API for running multiple external processes and multiplexing their output.

The current version of Honcho is 1.0.1 and it can be downloaded from GitHub or installed using pip: see *Installing Honcho*.

CHAPTER 1

Documentation index

Using Procfiles

As described in *What are Procfiles?*, Procfiles are simple text files that describe the components required to run an application. This document describes some of the more advanced features of Honcho and the Procfile ecosystem.

Syntax

The basic syntax of a Procfile is described in the Heroku Procfile documentation. In summary, a Procfile is a plain text file placed at the root of your applications source tree that contains zero or more lines of the form:

```
<process type>: <command>
```

The `process type` is a string which may contain alphanumerics and underscores (`[A-Za-z0-9_]+`), and uniquely identifies one type of process which can be run to form your application. For example: `web`, `worker`, or `my_process_123`.

`command` is a shell commandline which will be executed to spawn a process of the specified type.

Environment files

You can also create a `.env` file alongside your Procfile which contains environment variables which will be available to all processes started by Honcho:

```
$ cat >.env <<EOF
RACK_ENV=production
ASSET_ROOT=https://myapp.s3.amazonaws.com/assets
PROCFILE=Procfile
EOF
```

In addition to the variables specified in your `.env` file, the subprocess environment will also contain a `HONCHO_PROCESS_NAME` variable that will be set to a unique string composed of the process name as defined

in the Procfile and an integer counter that is incremented for each concurrent process of the same type, for example: `web.1`, `web.2`, `queue.1`, etc.

As shown, you may choose to specify your Procfile in the `.env` file. This takes priority over the default Procfile, but you can still use `-f` to replace which Procfile to use.

Typically, you should not commit your `.env` file to your version control repository, but you might wish to create a `.env.example` so that others checking out your code can see what environment variables your application uses.

For more on why you might want to use environment variables to configure your application, see Heroku's article on configuration variables and The Twelve-Factor App's guidance on configuration.

Using Honcho

To see the command line arguments accepted by Honcho, run it with the `--help` option:

```
$ honcho --help
usage: honcho [-h] [-e ENV] [-d DIR] [--no-colour] [--no-prefix] [-f FILE]
              [-v]
              {check,export,help,run,start,version} ...

Manage Procfile-based applications

optional arguments:
  -h, --help            show this help message and exit
  -e ENV, --env ENV     environment file[,file] (default: .env)
  -d DIR, --app-root DIR
                        procfile directory (default: .)
  --no-colour           disable coloured output
  --no-prefix           disable logging prefix
  -f FILE, --procfile FILE
                        procfile path (default: Procfile)
  -v, --version         show program's version number and exit

tasks:
  {check,export,help,run,start,version}
    check               validate a Procfile
    export              export a Procfile to another format
    help                describe available tasks or one specific task
    run                 run a command using your application's environment
    start               start the application (or a specific PROCESS)
    version             display honcho version
```

You will notice that by default, Honcho will read a Procfile called `Procfile` from the current working directory, and will read environment from a file called `.env` if one exists. You can override these options at the command line if necessary. For example, if your application root is a level above the current directory and your Procfile is called `Procfile.dev`, you could invoke Honcho thus:

```
$ honcho -d .. -f Procfile.dev start
16:14:49 web.1 | started with pid 1234
...
```

If you supply multiple comma-separated arguments to the `-e` option, Honcho will merge the environments provided by each of the files:

```
$ echo 'ANIMAL_1=giraffe' >.env.one
$ echo 'ANIMAL_2=elephant' >.env.two
$ honcho -e .env.one,.env.two run sh -c 'env | grep -i animal'
```

```
ANIMAL_1=giraffe
ANIMAL_2=elephant
```

Differences to Foreman

One of the curses of maintaining a "clone" of someone else's program is that you are forever asked to reimplement whatever questionable features upstream has introduced. So, while Honcho is based heavily on the Foreman project, there are some important differences between the two tools, some of which are simply the result of differences between Ruby and Python, and others are matters of software design. The following is a non-exhaustive list of these differences:

No *honcho run {target}*

Foreman allows you to specify a Procfile target to both the *start* and *run* subcommands. To me, it seems obvious that this functionality belongs only in *honcho start*, a command that always reads the Procfile and has no other use for its ARGV, as opposed to *honcho run*, which is intended for running a shell command in the environment provided by Honcho and *.env* files. Because I don't have to guess at whether or not ARGV is a process name or a shell command, *honcho start* even supports multiple processes: *honcho start web worker*.

Buffered output

By default, Python will buffer a program's output more aggressively than Ruby when a process has `STDOUT` connected to something other than a TTY. This can catch people out when running Python programs through Honcho: if the program only generates small amounts of output, it will be buffered, unavailable to Honcho, and will not display.

One way around this is to set the `PYTHONUNBUFFERED` environment variable in your `Procfile` or your `.env` file. Be sure you understand the performance implications of unbuffered I/O if you do this.

For example:

```
myprogram: PYTHONUNBUFFERED=true python myprogram.py
```

Exporting

Honcho allows you to export your Procfile configuration into other formats. Basic usage:

```
$ honcho export FORMAT LOCATION
```

Exporters for `upstart` and `supervisord` formats are shipped with Honcho.

Examples

The following command will create a `myapp.conf` file in the `/etc/supervisor/conf.d` directory:

```
$ honcho export -a myapp supervisord /etc/supervisor/conf.d
```

Or, for the upstart exporter:

```
$ honcho export -a myapp upstart /etc/init
```

By default, one of each process type will be started. You can change this by specifying the `--concurrency` option to `honcho export`.

Adding support for new export formats

You can add support for new export formats by writing plugins. Honcho discovers export plugins with the entry points mechanism of setuptools. Export plugins take the form of a class with `render` and `get_template_loader` methods that inherits from *honcho.export.base.BaseExport*. Inside the *render()* method, you can fetch templates using the *~honcho.export.base.BaseExport.get_template* method.

For example, here is a hypothetical exporter that writes out simple shell scripts for each process:

```
import jinja2

from honcho.export.base import BaseExport

class SimpleExport(BaseExport):
    def get_template_loader(self):
        return jinja2.PackageLoader(package_name=__package__,
                                    package_path='templates')

    def render(self, processes, context):
        tpl = get_template('run.sh')

        for p in processes:
            filename = 'run-{0}.sh'.format(p.name)
            ctx = context.copy()
            ctx['process'] = p
            script = tpl.render(ctx)
```

By writing an exporter in this way (specifically, by inheriting *BaseExport*), you make it possible for users of your exporter to override the exporter's default templates using the `--template-dir` option to `honcho export`.

In order for your export plugin to be detected by Honcho, you will need to register your exporter class under the `honcho_exporters` entrypoint. If we were shipping our hypothetical `SimpleExport` class in a package called `honcho_export_simple`, our `setup.py` might look something like the following:

```
from setuptools import setup

setup(
    name='honcho_export_simple',
    ...
    entry_points={
        'honcho_exporters': [
            'simple=honcho_export_simple:SimpleExport',
        ],
    },
)
```

After installing the package, the new export format will be shown by the `honcho export` command.

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

Types of Contributions

Report Bugs

Report bugs at https://github.com/nickstenning/honcho/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "feature" is open to whoever wants to implement it.

Write Documentation

Honcho could always use more documentation, whether as part of the official honcho docs, in docstrings, or even on the web in blog posts, articles, and such.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/nickstenning/honcho/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

Get Started!

Ready to contribute? Here's how to set up *honcho* for local development.

1. Fork the *honcho* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/honcho.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv honcho
$ cd honcho/
$ pip install -e .[export] tox
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass the tests, including testing other Python versions with tox and just run:

```
$ tox
```

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function or class with a docstring.
3. The pull request should work for Python 2.6, 2.7, 3.2 and 3.3 and for PyPy. Check https://travis-ci.org/nickstenning/honcho/pull_requests and make sure that the tests pass for all supported Python versions.

Tips

If you'd like to run a specific tox environment just use `-e` flag e.g.:

```
tox -e py27
```

This will run tests using python2.7 interpreter.

To list all available tox environments run:

```
tox -l
```

Honcho's tox setup uses pytest to run the test suite. You can pass positional arguments to a pytest command within tox. For example, if you'd like to use pytest's `-x` flag (stop after first error) with a PyPy interpreter you could do this:

```
tox -e pypy -- -x
```

API Documentation

class honcho.process.**Popen**(*cmd*, ***kwargs*)
Bases: subprocess.Popen

class honcho.process.**Process**(*cmd*, *name=None*, *colour=None*, *quiet=False*, *env=None*, *cwd=None*)
Bases: object

A simple utility wrapper around a subprocess.Popen that stores a number of attributes needed by Honcho and supports forwarding process lifecycle events and output to a queue.

run(*events=None*, *ignore_signals=False*)

class honcho.environ.**Env**
Bases: object

kill(*pid*)

now()

terminate(*pid*)

class honcho.environ.**ProcessParams**(*name*, *cmd*, *quiet*, *env*)
Bases: tuple

cmd
Alias for field number 1

env
Alias for field number 3

name
Alias for field number 0

quiet
Alias for field number 2

class honcho.environ.**Procfile**
Bases: object

A data structure representing a Procfile

add_process(*name*, *command*)

honcho.environ.**expand_processes**(*processes*, *concurrency=None*, *env=None*, *quiet=None*, *port=None*)
Get a list of the processes that need to be started given the specified list of process types, concurrency, environment, quietness, and base port number.

Returns a list of ProcessParams objects, which have *name*, *cmd*, *env*, and *quiet* attributes, corresponding to the parameters to the constructor of *honcho.process.Process*.

honcho.environ.**parse**(*content*)
Parse the content of a .env file (a line-delimited KEY=value format) into a dictionary mapping keys to values.

honcho.environ.**parse_procfile**(*contents*)

class honcho.manager.**Manager**(*printer=None*)
Bases: object

Manager is responsible for running multiple external processes in parallel managing the events that result (starting, stopping, printing). By default it relays printed lines to a printer that prints to STDOUT.

Example:

```
import sys
from honcho.manager import Manager

m = Manager()
m.add_process('server', 'ruby server.rb')
m.add_process('worker', 'python worker.py')
m.loop()

sys.exit(m.returncode)
```

add_process(*name*, *cmd*, *quiet=False*, *env=None*, *cwd=None*)
Add a process to this manager instance. The process will not be started until *loop()* is called.

kill()
Kill all processes managed by this ProcessManager.

loop()
Start all the added processes and multiplex their output onto the bound printer (which by default will print to STDOUT).

If one process terminates, all the others will be terminated by Honcho, and *loop()* will return.

This method will block until all the processes have terminated.

returncode = None

terminate()
Terminate all processes managed by this ProcessManager.

class honcho.export.base.**BaseExport**(*template_dir=None*, *template_env=None*)
Bases: object

get_template(*path*)
Retrieve the template at the specified path. Returns an instance of Jinja2.Template by default, but may be overridden by subclasses.

get_template_loader()

render(*processes*, *context*)

class honcho.export.base.**File**(*name*, *content*, *executable=False*)
Bases: object

honcho.export.base.**dashrepl**(*value*)
Replace any non-word characters with a dash.

honcho.export.base.**percentescape**(*value*)
Double any % signs.

class honcho.export.supervisord.**Export**(*template_dir=None*, *template_env=None*)
Bases: *honcho.export.base.BaseExport*

get_template_loader()

render(*processes*, *context*)

class honcho.export.upstart.**Export**(*template_dir=None*, *template_env=None*)
Bases: *honcho.export.base.BaseExport*

get_template_loader()

render(*processes*, *context*)

Credits

Development Lead

- Nick Stenning <nick@whiteink.com>

Contributors

- Jannis Leidel <jannis@leidel.info>
- Marc Abramowitz (@msabramo) <marc@marc-abramowitz.com> - maintainer
- Sławomir Ehlert (@slafs) <slafs.e@gmail.com> - maintainer
- Jiangge Zhang
- Thomas Orozco
- Alex Morega
- Igor Davydenko
- Alen Mujezinovic
- Jean-Philippe Serafin
- Alejandro Varas
- Hannes Struss
- Jeethu Rao
- Jökull Sólberg Auðunsson
- Andrii Kurinnyi
- Chad Whitacre
- Hyunjun Kim
- Jesse Pollak
- Mark Burnett
- Miguel Grinberg
- Pepijn de Vos
- Philippe Ombredanne

CHAPTER 2

What are Procfiles?

A Procfile is a file which describes how to run your application. If you need to run a simple web application, you might have a Procfile that looks like this:

```
web: python myapp.py
```

You'd then be able to run your application using the following command:

```
$ honcho start
```

Now, if running your application is as simple as typing `python myapp.py`, then perhaps Honcho isn't that useful. But imagine that a few months have passed, and running your application is now substantially more complicated. You need to have the following running in parallel: a web server, a high priority job queue worker, and a low priority job queue worker. In addition, you've established that you need to run your application under a proper web server like gunicorn. Now the Procfile starts to be useful:

```
web: gunicorn -b "0.0.0.0:$PORT" -w 4 myapp:app
worker: python worker.py --priority high,med,low
worker_low: python worker.py --priority med,low
```

Again, you can start all three processes with a single command:

```
$ honcho start
```

As you add features to your application, you shouldn't be forced to bundle everything up into a single process just to make the application easier to run. The Procfile format allows you to specify how to run your application, even when it's made up of multiple independent components. Honcho (and Foreman, and Heroku) can parse the Procfile format and run your application.

CHAPTER 3

Why did you port Foreman?

Foreman is a great tool, and the fact I chose to port it to Python shouldn't be interpreted as saying anything negative about Foreman. But I've worked in Python-only development environments, where installing Ruby just so I can run Procfile applications seemed a bit crazy. Python, on the other hand, is part of the Linux Standard Base, and so even in "Ruby-only" environments, Python will still be around.

(Oh, and I also I wanted to learn about asynchronous I/O in Python.)

CHAPTER 4

Installing Honcho

If you have a working Python and pip installation, you should be able to simply

```
pip install honcho
```

and get a working installation of Honcho. You can probably also `easy_install honcho`. But please, don't: get with the program.

CHAPTER 5

Further reading and assistance

For more about the Procfile format, `.env` files, and command-line options to Honcho, see *Using Procfiles*.

If you have any difficulty using Honcho or this documentation, please get in touch with me, Nick Stenning, on Twitter at @nickstenning or by email: `<my first name> at whiteink dot com`.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

h

Index