parcel Documentation

Release 0.2

Crispin Wellington

April 18, 2014

Contents

CHAPTER 1

Parcel: Pragmatic Python Webapp Deployment

Parcel v0.2. (*Installation*)

Parcel is a suite of classes and helper functions designed to be used in conjunction with Fabric for building and deploying Python web applications using native packages.

CHAPTER 2

Feature Support

Although Parcel is intended to be fairly minimal, it presently supports:

- Debian targets
- CentOS targets
- uWSGI deployment
- Git and Mercurial source trees

CHAPTER 3

Introduction

An overview of Parcel's origins and intended uses.

3.1 Origin

Hynek Schlawack wrote an article about python application deployment using native linux packages. This project is an attempt to formalise some of those ideas into a more generic installation framework that follows those guidelines.

3.2 Build Host

Parcel uses the concept of a *build host* (or hosts) to simplify build procedures. The premise is thus: for each platform you want to target for deployment, you have a build host of the same architecture with the same software on it. Fabric connects to these machines and issues the build commands. This way the binary package created will be tailored to the final hardware.

These build hosts can be real machines, virtual machines, or cloud instances. It is important that the build hosts mirror your live deployment environment.

So for example, if your live servers were a mixture of Redhat on 64bit Intel architecture, and Debian on 32bit hardware, then you would have two build hosts, one Redhat 64bit and one Debian 32bit. You would then build rpms and debs of your project on those respective boxes.

The build hosts are specified to Fabric via the -H hosts command. For example the following will build two deb packages for the project for two different platforms:

```
$ fab -H debian32.localdomain,debian64.localdomain deb
```

3.2.1 Localhost as Hub

Parcel conforms to the idea of being the hub in all connections. Where everything is being pushed to and pulled from the local machine from which you issue your *fab* commands. So for instance, after a .deb is built, it is pulled back to the localhost. When the build host needs a new version of code, the code is copied across to the build machine. It is *not* checked out on the build machine. When completed packages are pushed into repositories, it is done from the local machine, not from the build host.

This has the advantages of keeping all connection authentication local to the developers machine. No private credentials need to be moved onto a build host to enable it to connect further. Once you have an ssh agent with your keys installed everything runs seamlessly.

It also helps keep the software that needs to be installed on the build host to a minimum. For example, mercurial or git does not need to be installed on the build host.

3.2.2 Build From Source, Deploy As Binary

Parcel performs builds on the build host from source rather than binaries. Then that source is honed down into a package that is distributed as a binary. Rather than builds being performed using eggs, or packages, all the source is checked out and a full source build is done. This ensures complete compatability with the deployment architecture and system arangement. Once this full source build is done, the resultant compiled files are packaged.

3.3 Parcel License

CHAPTER 4

Installation

Installation of Parcel.

4.1 Installing with Pip

You can install Parcel with Pip:

```
$ pip install parcel
```

4.2 Installing from source

Parcel is available on the BitBucket project page.

Clone the public repository:

```
$ hg clone https://bitbucket.org/andrewmacgregor/parcel
```

Once you have a copy of the source code, install it into your python environment:

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

4.3 Checking Parcel is installed

To check that Parcel is successfully installed:

```
$ python
Python 2.6.1 (r261:67515, Jun 24 2010, 21:47:49)
[GCC 4.2.1] on debian
Type "help", "copyright", "credits" or "license" for more information.
>>> import parcel
```

CHAPTER 5

Quickstart

A quick introduction to Parcel for those who want to dive on in.

5.1 Quickstart

This page gives a good introduction to getting started with Parcel. This assumes you already have Parcel installed. If you do not, head over to the *Installation* section.

You will also need to have a build host set up that you can use to build some packages. You could setup a *Debian machine to be a build host* for this quickstart guide.

From here on in we assume that your build host is a Debian machine available with the name *debian.localdomain*. Replace that name where it occurs below with your own Debian machine's hostname or IP address. Let's get started with some simple examples.

5.1.1 Making a Package

Making a package is very simple. Begin by going to the base directory of your project and making a file *fabfile.py*.

Then in that file write the following:

```
from parcel.deploy import Deployment

env.app_name = "myapp"

@task
def deb():
    deploy = Deployment(env.app_name)
    deploy.prepare_app()
    deploy.build_package()
```

Now save the fabfile and at the commandline issue:

```
$ fab -H debian.localdomain deb
```

When the build is finished you should have a file *myapp_0.0.1_all.deb*:

```
$ ls -l *.deb
```

5.1.2 Package Details

If you want to see what's been put in the package, use the deb_ls target found in parcel.probes. First add the following to *fabfile.py*:

```
from parcel.probes import *
```

Then you can use deb_ls to list the contents of the package:

```
$ fab -H debian.localdomain deb_ls:myapp_0.0.1_all.deb
```

You will see that the package consists of all the files in your source directory. This is the simplest form of packaging. This is not that useful as it is only the files. But from here your fabfile can expand to implement some deployment scenarios.

If you look at the packages control files with:

```
$ fab -H debian.localdomain deb_control:myapp_0.0.1_all.deb
```

you will notice the package we have built contains no install or remove scripts. You can also see a filesystem tree of the final installed package with:

```
$ fab -H debian.localdomain deb_tree:myapp_0.0.1_all.deb
```

CHAPTER 6

Tutorials

A more in-depth guide to using Parcel to build and deploy packages for common web applications.

6.1 Tutorials

Heres some step by step tutorials of deploying small example projects in popular python web frameworks.

6.1.1 Django

Setting Up The Project

Start with an empty directory. Make a *requirements.txt* file and put the following in it:

```
django==1.5
```

Let's build a virtual python for our development:

```
$ virtualenv vp
$ vp/bin/pip install -r requirements.txt
```

This should download and install django locally. Let's activate our virtual python and start a django project:

```
$ source vp/bin/activate
$ django-admin.py startproject myproject
```

Let's change into the project directory and fire up the development web server:

```
$ cd myproject/
$ python manage.py runserver
```

Point your browser at http://localhost:8000/ and see the django splash page.

Writing The Fabfile

Go back to the project base. That's the directory with the virtual python 'vp' directory one above 'myproject':

```
$ cd ..
```

Make a *fabfile.py* here and put the following in it:

```
from parcel.deploy import uWSGI
from parcel.probes import *

from fabric.api import task

@task
def deb():
    deploy = uWSGI("myproject", base="webapps")
    deploy.prepare_app()
    deploy.add_supervisord_uwsgi_service("mproject",port=10000)
    deploy.build_deb()
```

Let's take a look at what is in this file. First thing is that we are using the uWSGI Deployment object. This is a stripped down uwsgi deployment container.

The first argument to Deployment is the package name. This will be used to name the binary package when it's built.

The variable *base* is a path on the remote host that the package will be installed into. If this is ommitted, the install path is the home directory of the building user. If it's an absolute path (starting with /) then it's installed in that path. If it's a relative path like this setting, it is installed into that path relative to the build users home directory. So in our case it will be "~/webapps". So, for instance, if we were to build the package as user apache (pass *-u apache* into our fab call), then the package on debian would be installed under /var/www/webapps.

The *prepare_app* call copies the code over to the build host and setsup the virtual env with the requirements.txt file. Then the *add_supervisord_uwsgi_service* sets the package to make a uwsgi daemon start under supervisord. This daemon will listen for web requests on port 10000 and the supervisor service is called 'myproject'. The final line builds the deb.

Go ahead and build the package:

```
$ fab -H debian deb
```

You'll notice as the files are copied they are listed in blue. In this list you will see that there are some files in your directory you don't want in there. Firstly, you don't want any local virtualenv being copied over, as the package builds its own on the destination target. You also don't want your fabfile copied over. Once you build the deb, if you run the build again, *the deb file you just created would be copied over*! So lets ignore these by listing their globs in a file called *.rsync-ignore*. You also want to ignore this file.

Edit *.rsync-ignore* and put in it the following:

```
vp
*.deb
fabfile*
.rsync-ignore
```

Now build the deb again and notice that these files are excluded from the copy:

```
$ fab -H debian deb
```

Once it's built, you can have a look at directory structure:

```
$ fab -H debian deb_tree:myproject_0.0.0_all.deb
```

and more interestingly the package's control files:

```
$ fab -H debian deb_control:myproject_0.0.0_all.deb
```

Now its time to test whether the package will install. Install the package on the build host as root:

```
$ fab -H debian -u root deb_install:myproject_0.0.0_all.deb
```

Please note that this install method uses dpkg and therefore requires run dependencies to be installed before calling *deb_install*.

Now point your browser at http://debian:10000/ (where debian is the hostname/ip where you installed the deb package) and you should see exactly what you saw from runserver locally.

Congratulations! You just packaged and deployed a Django application.

CHAPTER 7

Cookbook

Common recipes to use Parcel in your fabfiles.

7.1 Cookbook

Here are common recipes that you may use with Parcel. All of these code samples are available in the docs/examples directory.

7.1.1 Building a Debian Package

```
from fabric.api import env, task
from fabric.tasks import Task
from parcel import distro
from parcel import deploy

env.app_name = 'default_app_name'
env.run_deps = []
env.build_deps = []
env.base = None
env.path = '.'
env.arch = distro.debian

@task
def build():
    """Instantiate a Deployment object and
    build a Debian based package."""
    d = deploy.Deployment(app_name=env.app_name,
                          build_deps=env.build_deps,
                          run_deps=env.run_deps,
                          path=env.path,
                          base=env.base,
                          arch=env.arch
                          )
    d.prepare_app()
    d.build_package()
```

7.1.2 Building a Debian Package for uWSGI deployment

```
from fabric.api import env, task
from fabric.tasks import Task
from parcel import distro
from parcel import deploy

env.app_name = 'default_app_name'
env.run_deps = []
env.build_deps = []
env.base = None
env.path = '.'
env.arch = distro.Debian()

@task
def build_for_uwsgi():
    """Instantiate a Deployment object and build a Debian based package for
    uwsgi deployment."""
    assert hasattr(env, 'service_name'), "You need to set env.service_name"
    assert hasattr(env, 'service_port'), "You need to set env.service_port"
    d = deploy.uWSGI(app_name=env.app_name,
                     build_deps=env.build_deps,
                     run_deps=env.run_deps,
                     path=env.path,
                     base=env.base,
                     arch=env.arch
                     )
    d.prepare_app()
    d.add_supervisord_uwsgi_service(env.service_name, env.service_port)
    d.build_package()
```

7.1.3 Building a Centos RPM Package

```
from fabric.api import env, task
from fabric.tasks import Task
from parcel import distro
from parcel import deploy
from parcel.helpers import setup_centos

# NB: Centos will need to have EPEL activated and be using a more modern python than 2.4

env.app_name = 'default_app_name'
env.run_deps = []
env.build_deps = []
env.base = "/usr/local/webapps/"
env.path = '.'
env.arch = distro.centos

@task
def build():
    """Instantiate a Deployment object and
    build an RPM based package."""
    d = deploy.Deployment(app_name=env.app_name,
                          build_deps=env.build_deps,
                          run_deps=env.run_deps,
                          path=env.path,
                          base=env.base,
```

```
                        arch=env.arch
                        )

    d.prepare_app()
    d.build_package()
```

CHAPTER 8

Build Host

How to set up a machine to act as a build host for Parcel.

8.1 Making a Build Host

Start with a new *fabfile.py* and in it put the following single line:

```
from parcel.helpers import *
```

This will bring in a bunch of standard fab targets for helping to setup build hosts. Run a *fab -l* to see the targets:

```
$ fab -l
Available commands:

    copy_ssh_key  This copies the local uses id_rsa.pub and id_dsa.pub keys into the authorized_keys
    setup_debian  Set up the build host for building in a debian way
```

If your key is not already on the build host, copy it across with copy_ssh_key:

```
$ fab -H hostname -u root copy_ssh_key
```

Put in the root password when asked and this will set you up for passwordless ssh root access.

8.1.1 Debian

To setup a debian build host, use the setup_debian target:

```
$ fab -H hostname -u root setup_debian
```

CHAPTER 9

API

9.1 API

This part of the documentation covers the application programming interface of Parcel.

9.1.1 Deployment

The majority of work done by Parcel is using the `Deployment` object.

class `parcel.deploy.`**`Deployment`**(*app_name*, *build_deps=None*, *run_deps=None*, *path='.'*, *base=None*, *arch=<parcel.distro.Debian object at 0x37b2b50>*, *version=None*, *venv_dirname='vp'*)

The core `Deployment` object. All Fabric tasks built with Parcel will probably use this an instance of this class.

`add_data_to_root_fs`(*data*, *remotepath*)
Copies data in file on remotepath (relative to final root)

`add_postinst`(*lines*)
Add lines to the postinst file

`add_postrm`(*lines*)
Add lines to the postrm file

`add_preinst`(*lines*)
Add lines to the preinst file

`add_prerm`(*lines*)
Add lines to the prerm file

`add_to_root_fs`(*localfile*, *remotepath*)
Add a local file to the root package path. If remote path ends in /, the filename is copied into that directory. If the remote path doesn't end in /, it represents the final filename.

`app_name` = **None**
The name of the resulting package.

`arch` = **None**
The architecture of the build host. This should be a `Distro` object.

`base_path` = **None**
Location of files during build on build host. Default is user's home directory. If path is relative, it's relative to the remote user's home directory. If the path is absolute, it's used as is.

build_deps = None
A list of packages that need to be installed to build the software.

build_package(*templates=True*)
Takes the whole app including the virtualenv, packages it using fpm and downloads it to the local host. The version of the package is the build number - which is just the latest package version in our Ubuntu repositories plus one.

path = None
The directory that will be used as the base level directory.

prepare_app(*branch=None*, *requirements='requirements.txt'*)
Creates the necessary directories on the build server, checks out the desired branch (None means current), creates a virtualenv and populates it with dependencies from requirements.txt.

Parameters **requirements** – The name of the requirements.txt file relative to the path setting used in the constructor.

run_deps = None
A list of packages that must be installed to run the resulting package.

write_postinst_template(*template*)
Take a template postinst script and format it with appname and postinst_lines If you call this function you must supply a template string that includes {app_name} and {lines}.

write_postrm_template(*template*)
Take a template postrm script and format it with appname and postrm_lines If you call this function you must supply a template string that includes {app_name} and {lines}.

write_preinst_template(*template*)
Take a template preinst script and format it with appname and preinst_lines If you call this function you must supply a template string that includes {app_name} and {lines}.

write_prerm_template(*template*)
Take a template prerm script and format it with appname and prerm_lines If you call this function you must supply a template string that includes {app_name} and {lines}.

9.1.2 Hg

Access your Mercurial revision control information with the `Hg` objects.

class parcel.revisions.**Hg**(*path*)
An interface to Mercurial source repositories.

branch()
A property that is the present checked out branch

clone(*repo*)
hg clone a repo or path to the present repo location. The Hg path and object this is called on should be clean. In other words you should call clone() immedately after construction of the Hg object and make sure that the Hg object is constructed on an empty path.

eg.

hg = Hg("build/clone") hg.clone("hg+ssh://bitbucet.org/project")

describe(*template='{latesttag}-{latesttagdistance}-{node|short}'*)
Create a vesrion tag composed of the latest tag, the tag distance, and the short hash. For example:

0.5.1-23-d63d252639de

composed of the tag 0.5.1, from which we are 23 commits forwards of, with a latest changeset of hash d63d252639de

log()
A property that is the latest log entry. Returns a dictionary with the following keys: changeset: The hash number of the lastest changeset. date: The date and time of the latest changeset. user: Who committed the change summary: The commit message tag: if this commit is tagged, this is the tag.

logs()
Returns all the log entries as a list of dictionaries. Each dictionary is of the format returned by log.

path = None
The base path of the Mercurial repository.

pull()
Issue a hg pull on the repository

update()
Issue a hg update on the repository

9.1.3 Git

Access your Git revision control information with the `Git` objects.

class parcel.revisions.**Git**(*path*)
An abstraction of Git Repositories.

branch()
Return which branch the repository is checked out on.

checkout(*target*)
Use git checkout to bring the repository to a particular point

clone(*repo*)
git clone a repo or path to the present repo location. The Git path and object this is called on should be clean. In other words you should call clone() immedately after construction of the Git object and make sure that the Git object is constructed on an empty path.

eg.

git = Git("build/clone") git.clone("git://github.org/project")

describe(*force_tag=False*)
Use git describe to create a version tag describing the repository at this point.

log()
Return all the git logs in a list

pull()
Execute a git pull in the repository

9.1.4 Distro

Code specific to different distributions can be found in the `Distro` object.

class parcel.distro.**Distro**(*use_sudo=False*)
The base class for Distro classes. If use_sudo is true, then super user commands will be run using fabric's sudo call. If sudo is false, super user access is gained by getting fabric to connect as root user.

build_deps(*deps*)
This method should install the build dependencies on the remote box.

check()
Check the remote build host to see if the relevant software to build packages is installed

install_package(*pkg*)
Installs package on the host using apt-get install bypassing authentication. This method should be used for testing package installation before using push_to_repo.

mkdir(*remote*)
Make a directory on the remote

push_files(*pathlist*, *dst*)
Push all the files in pathlist into dst directory on remote.

setup()
This method should set up a remote box for parcel package building. It should install fpm.

su(**args*, ***kwargs*)
Method to perform a remote task as a super user. Takes same arguments as fabric.api.run or fabric.api.sudo. Can be overridden to provide your own super user execution hook.

update_packages()
This method should update the packages on the remote box.

version(*package*)
Look at the distro's packaging system for the package and return a version

9.1.5 Debian

Code specific to the Debian distribution can be found in the `Debian` object.

class parcel.distro.**Debian**(**args*, ***kwargs*)

build_package(*deployment=None*)
Runs architecture specific packaging tasks

check()
Check the remote build host to see if the relevant software to build packages is installed

install_package(*pkg*)
Installs package on the host using apt-get install bypassing authentication. This method should be used for testing package installation before using push_to_repo.

mkdir(*remote*)
Make a directory on the remote

push_files(*pathlist*, *dst*)
Push all the files in pathlist into dst directory on remote.

setup()
this method sets up a remote debian box for parcel package building. Installs fpm, easyinstall and some libraries.

su(**args*, ***kwargs*)
Method to perform a remote task as a super user. Takes same arguments as fabric.api.run or fabric.api.sudo. Can be overridden to provide your own super user execution hook.

version(*package*)
Look at the debian apt package system for a package with this name and return its version. Return None if there is no such package.

9.1.6 Ubuntu

Code specific to the Ubuntu distribution can be found in the `Ubuntu` object.

class `parcel.distro.`**Ubuntu**(**args*, ***kwargs*)

build_package(*deployment=None*)
Runs architecture specific packaging tasks

check()
Check the remote build host to see if the relevant software to build packages is installed

install_package(*pkg*)
Installs package on the host using apt-get install bypassing authentication. This method should be used for testing package installation before using push_to_repo.

mkdir(*remote*)
Make a directory on the remote

push_files(*pathlist*, *dst*)
Push all the files in pathlist into dst directory on remote.

setup()
this method sets up a remote ubuntu box for parcel package building. Installs fpm and also rubygems if not present.

su(**args*, ***kwargs*)
Method to perform a remote task as a super user. Takes same arguments as fabric.api.run or fabric.api.sudo. Can be overridden to provide your own super user execution hook.

version(*package*)
Look at the debian apt package system for a package with this name and return its version. Return None if there is no such package.

9.1.7 CentOS

Code specific to the CentOS distribution can be found in the `Centos` object.

class `parcel.distro.`**Centos**(**args*, ***kwargs*)

build_package(*deployment=None*)
Runs architecture specific packaging tasks

check()
Check the remote build host to see if the relevant software to build packages is installed

install_package(*pkg*)
Installs package on the host using apt-get install bypassing authentication. This method should be used for testing package installation before using push_to_repo.

mkdir(*remote*)
Make a directory on the remote

push_files(*pathlist*, *dst*)
Push all the files in pathlist into dst directory on remote.

setup()
this method sets up a remote centos box for parcel package building. Installs fpm and also rubygems if not present.

su(**args*, ***kwargs*)
Method to perform a remote task as a super user. Takes same arguments as fabric.api.run or fabric.api.sudo. Can be overridden to provide your own super user execution hook.

version(*package*)
Look at the debian apt package system for a package with this name and return its version. Return None if there is no such package.

9.1.8 Helpers

Various helper functions are available in parcel.helpers

parcel.helpers.**copy_ssh_key**()

parcel.helpers.**setup_debian**()

parcel.helpers.**setup_ubuntu**()

parcel.helpers.**setup_centos**()

9.1.9 Probes

Functions for inspecting built packages are available in parcel.probes

parcel.probes.**deb_ls**()

parcel.probes.**deb_install**()

parcel.probes.**deb_control**()

parcel.probes.**deb_tree**()

CHAPTER 10

Authors

10.1 Authors

Parcel is written and maintained by Crispin Wellington and Andrew Macgregor:

10.1.1 Development Lead

- Crispin Wellington <retrogradeorbit@gmail.com>

10.1.2 Core Developers

- Crispin Wellington <retrogradeorbit@gmail.com>
- Andrew Macgregor <amacgregor@gmail.com>

10.1.3 Patches and Suggestions

- Michael Heyvaert
- Josh Hansen

CHAPTER 11

Indices and tables

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

p