Paste Deploy

Release 1.5.2

June 07, 2017

Contents

author Ian Bicking <ianb@colorstudy.com>

Contents

Documents:

CHAPTER 1

Paste Deployment News

1.5.2

- Fixed Python 3 issue in paste.deploy.util.fix_type_error()

1.5.1

- Fixed use of the wrong variable when determining the context protocol
- Fixed invalid import of paste.deploy.Config to paste.deploy.config.Config
- Fixed multi proxy IPs bug in X-Forwarded-For header in PrefixMiddleware
- Fixed TypeError when trying to raise LookupError on Python 3
- Fixed exception reraise on Python 3

Thanks to Alexandre Conrad, Atsushi Odagiri, Pior Bastida and Tres Seaver for their contributions.

1.5.0

- Project is now maintained by Alex Grönholm <alex.gronholm@nextday.fi>
- Was printing extraneous data when calling setup.py
- Fixed missing paster template files (fixes "paster create -t paste.deploy")
- Excluded tests from release distributions
- Added support for the "call:" protocol for loading apps directly as functions (contributed by Jason Stitt)
- Added Python 3.x support
- Dropped Python 2.4 support

- Removed the `paste.deploy.epdesc` and `paste.deploy.interfaces` modules – contact the maintainer if you actually needed them

1.3.4

- Fix loadconfig path handling on Jython on Windows.

1.3.3

- In `paste.deploy.config.PrefixMiddleware` the headers `X-Forwarded-Scheme` and `X-Forwarded-Proto` are now translated to the key `environ['wsgi.url_scheme']`. Also `X-Forwarded-For` is translated to `environ['REMOTE_ADDR']`
- Also in PrefixMiddleware, if X-Forwarded-Host has multiple (comma-separated) values, use only the first value.

1.3.2

- Added `paste.deploy.converters.asint()`.
- fixed use sections overwriting the config's __file__ value with the use'd filename.
- `paste.deploy.loadwsgi` now supports variable expansion in the DEFAULT section of config files (unlike plain ConfigParser).

1.3.1

- Fix `appconfig` config loading when using a config file with `filter-with` in it (previously you'd get TypeError: iteration over non-sequence)

1.3

- Added `scheme` option to `PrefixMiddleware`, so you can force a scheme (E.g., when proxying an HTTPS connection over HTTP).
- Pop proper values into `environ['paste.config']` in `ConfigMiddleware`.

1.1

- Any `global_conf` extra keys you pass to `loadapp` (or the other loaders) will show up as though they were in `[DEFAULT]`, so they can be used in variable interpolation. Note: this won't overwrite any existing values in `[DEFAULT]`.
- Added `force_port` option to `paste.deploy.config.PrefixMiddleware`. Also the `prefix` argument is stripped of any trailing `/`, which can't be valid in that position.

1.0

- Added some documentation for the different kinds of entry points Paste Deploy uses.
- Added a feature to `PrefixMiddleware` that translates the `X-Forwarded-Server` header to `Host`.

0.9.6

- Added `PrefixMiddleware` which compensates for cases where the wsgi app is behind a proxy of some sort that isn't moving the prefix into the SCRIPT_NAME in advance.
- Changed _loadconfig() so that it works with Windows absolute paths.
- Make the error messages prettier when you call a function and fail to give an argument, like a required function argument.

0.5

- Made the `paste_deploy` template (used with `paster create --template=paste_deploy`) more useful, with an example application and entry point.

0.4

- Allow filters to have `filter-with` values, just like applications.
- Renamed `composit` to `composite` (old names still work, but aren't documented).
- Added `appconfig()` to load along with `loadapp()`, but return the configuration without invoking the application.

0.3

- Allow variable setting like:

```
get local_var = global_var_name
```

  To bring in global variables to the local scope.

- Allow interpolation in files, like `%(here)s`. Anything in the `[DEFAULTS]` section will be available to substitute into a value, as will variables in the same section. Also, the special value `here` will be the directory the configuration file is located in.

0.2

Released 26 August 2004

- Added a `filter-with` setting to applications.
- Removed the `1` from all the protocol names (e.g., `paste.app_factory1` is not `paste.app_factory`).

- Added `filter-app:` and `pipeline:` sections. Docs.
- Added `paste.filter_app_factory1` (doc) and `paste.server_runner1` (doc) protocols.
- Added `paste.deploy.converters` module for handling the string values that are common with this system.

0.1

Released 22 August 2004

Initial version released. It's all new.

CHAPTER 2

paste.deploy.loadwsgi – Load WSGI applications from config files

Module Contents

CHAPTER 3

paste.deploy.config – Configuration and Environment middleware

Module Contents

CHAPTER 4

paste.deploy.converters – Conversion helpers for String Configuration

Module Contents

CHAPTER 5

Introduction

Paste Deployment is a system for finding and configuring WSGI applications and servers. For WSGI application consumers it provides a single, simple function (`loadapp`) for loading a WSGI application from a configuration file or a Python Egg. For WSGI application providers it only asks for a single, simple entry point to your application, so that application users don't need to be exposed to the implementation details of your application.

The result is something a system administrator can install and manage without knowing any Python, or the details of the WSGI application or its container.

Paste Deployment currently does not require other parts of Paste, and is distributed as a separate package.

To see updates that have been made to Paste Deploy see the news file.

Paste Deploy is released under the MIT license.

CHAPTER 6

Status

Paste Deploy has passed version 1.0. Paste Script is an actively maintained project. As of 1.0, we'll make a strong effort to maintain backward compatibility (this actually started happening long before 1.0, but now it is explicit). This will include deprecation warnings when necessary. Major changes will take place under new functions or with new entry points.

Note that the most key aspect of Paste Deploy is the entry points it defines (such as `paste.app_factory`). Paste Deploy is not the only consumer of these entry points, and many extensions can best take place by utilizing the entry points instead of using Paste Deploy directly. The entry points will not change; if changes are necessary, new entry points will be defined.

CHAPTER 7

Installation

First make sure you have either setuptools or its modern replacement distribute installed. For Python 3.x you need distribute as setuptools does not work on it.

Then you can install Paste Deployment using pip by running:

```
$ sudo pip install PasteDeploy
```

If you want to track development, do:

```
$ hg clone http://bitbucket.org/ianb/pastedeploy
$ cd pastedeploy
$ sudo python setup.py develop
```

This will install the package globally, but will load the files in the checkout. You can also simply install `PasteDeploy==dev`.

For downloads and other information see the Cheese Shop PasteDeploy page.

A complimentary package is Paste Script. To install that, use `pip install PasteScript` (or `pip install PasteScript==dev`).

CHAPTER 8

From the User Perspective

In the following sections, the Python API for using Paste Deploy is given. This isn't what users will be using (but it is useful for Python developers and useful for setting up tests fixtures).

The primary interaction with Paste Deploy is through its configuration files. The primary thing you want to do with a configuration file is serve it. To learn about serving configuration files, see *the ‘‘paster serve‘* command <http://pythonpaste.org/script/#paster-serve>‘_.

The Config File

A config file has different sections. The only sections Paste Deploy cares about have prefixes, like `app:main` or `filter:errors` – the part after the `:` is the "name" of the section, and the part before gives the "type". Other sections are ignored.

The format is a simple INI format: `name = value`. You can extend the value by indenting subsequent lines. `#` is a comment.

Typically you have one or two sections, named "main": an application section (`[app:main]`) and a server section (`[server:main]`). `[composite:...]` signifies something that dispatches to multiple applications (example below).

Here's a typical configuration file that also shows off mounting multiple applications using paste.urlmap:

```
[composite:main]
use = egg:Paste#urlmap
/ = home
/blog = blog
/wiki = wiki
/cms = config:cms.ini

[app:home]
use = egg:Paste#static
document_root = %(here)s/htdocs
```

```
[filter-app:blog]
use = egg:Authentication#auth
next = blogapp
roles = admin
htpasswd = /home/me/users.htpasswd

[app:blogapp]
use = egg:BlogApp
database = sqlite:/home/me/blog.db

[app:wiki]
use = call:mywiki.main:application
database = sqlite:/home/me/wiki.db
```

I'll explain each section in detail now:

```
[composite:main]
use = egg:Paste#urlmap
/ = home
/blog = blog
/cms = config:cms.ini
```

That this is a `composite` section means it dispatches the request to other applications. `use = egg:Paste#urlmap` means to use the composite application named `urlmap` from the `Paste` package. `urlmap` is a particularly common composite application – it uses a path prefix to map your request to another application. These are the applications like "home", "blog", "wiki" and "config:cms.ini". The last one just refers to another file `cms.ini` in the same directory.

Next up:

```
[app:home]
use = egg:Paste#static
document_root = %(here)s/htdocs
```

`egg:Paste#static` is another simple application, in this case it just serves up non-dynamic files. It takes one bit of configuration: `document_root`. You can use variable substitution, which will pull variables from the section `[DEFAULT]` (case sensitive!) with markers like `%(var_name)s`. The special variable `%(here)s` is the directory containing the configuration file; you should use that in lieu of relative filenames (which depend on the current directory, which can change depending how the server is run).

Then:

```
[filter-app:blog]
use = egg:Authentication#auth
next = blogapp
roles = admin
htpasswd = /home/me/users.htpasswd

[app:blogapp]
use = egg:BlogApp
database = sqlite:/home/me/blog.db
```

The `[filter-app:blog]` section means that you want an application with a filter applied. The application being filtered is indicated with `next` (which refers to the next section). The `egg:Authentication#auth` filter doesn't actually exist, but one could imagine it logs people in and checks permissions.

That last section is just a reference to an application that you probably installed with `pip install BlogApp`, and one bit of configuration you passed to it (`database`).

Lastly:

```
[app:wiki]
use = call:mywiki.main:application
database = sqlite:/home/me/wiki.db
```

This section is similar to the previous one, with one important difference. Instead of an entry point in an egg, it refers directly to the `application` variable in the `mywiki.main` module. The reference consist of two parts, separated by a colon. The left part is the full name of the module and the right part is the path to the variable, as a Python expression relative to the containing module.

So, that's most of the features you'll use.

CHAPTER 9

Basic Usage

The basic way you'll use Paste Deployment is to load WSGI applications. Many Python frameworks now support WSGI, so applications written for these frameworks should be usable.

The primary function is `paste.deploy.loadapp`. This loads an application given a URI. You can use it like:

```
from paste.deploy import loadapp
wsgi_app = loadapp('config:/path/to/config.ini')
```

There's two URI formats currently supported: `config:` and `egg:`.

CHAPTER 10

config: URIs

URIs that being with `config:` refer to configuration files. These filenames can be relative if you pass the `relative_to` keyword argument to `loadapp()`.

Note: Filenames are never considered relative to the current working directory, as that is a unpredictable location. Generally when a URI has a context it will be seen as relative to that context; for example, if you have a `config:` URI inside another configuration file, the path is considered relative to the directory that contains that configuration file.

Config Format

Configuration files are in the INI format. This is a simple format that looks like:

```
[section_name]
key = value
another key = a long value
    that extends over multiple lines
```

All values are strings (no quoting is necessary). The keys and section names are case-sensitive, and may contain punctuation and spaces (though both keys and values are stripped of leading and trailing whitespace). Lines can be continued with leading whitespace.

Lines beginning with `#` (preferred) or `;` are considered comments.

Applications

You can define multiple applications in a single file; each application goes in its own section. Even if you have just one application, you must put it in a section.

Each section name defining an application should be prefixed with `app:`. The "main" section (when just defining one application) would go in `[app:main]` or just `[app]`.

There's two ways to indicate the Python code for the application. The first is to refer to another URI or name:

```
[app:myapp]
use = config:another_config_file.ini#app_name

# or any URI:
[app:myotherapp]
use = egg:MyApp

# or a callable from a module:
[app:mythirdapp]
use = call:my.project:myapplication

# or even another section:
[app:mylastapp]
use = myotherapp
```

It would seem at first that this was pointless; just a way to point to another location. However, in addition to loading the application from that location, you can also add or change the configuration.

The other way to define an application is to point exactly to some Python code:

```
[app:myapp]
paste.app_factory = myapp.modulename:app_factory
```

You must give an explicit *protocol* (in this case `paste.app_factory`), and the value is something to import. In this case the module `myapp.modulename` is loaded, and the `app_factory` object retrieved from it.

See *Defining Factories* for more about the protocols.

Configuration

Configuration is done through keys besides `use` (or the protocol names). Any other keys found in the section will be passed as keyword arguments to the factory. This might look like:

```
[app:blog]
use = egg:MyBlog
database = mysql://localhost/blogdb
blogname = This Is My Blog!
```

You can override these in other sections, like:

```
[app:otherblog]
use = blog
blogname = The other face of my blog
```

This way some settings could be defined in a generic configuration file (if you have `use = config:other_config_file`) or you can publish multiple (more specialized) applications just by adding a section.

Global Configuration

Often many applications share the same configuration. While you can do that a bit by using other config sections and overriding values, often you want that done for a bunch of disparate configuration values. And typically applications can't take "extra" configuration parameters; with global configuration you do something equivalent to "if this application wants to know the admin email, this is it".

Applications are passed the global configuration separately, so they must specifically pull values out of it; typically the global configuration serves as the basis for defaults when no local configuration is passed in.

Global configuration to apply to every application defined in a file should go in a special section named `[DEFAULT]`. You can override global configuration locally like:

```
[DEFAULT]
admin_email = webmaster@example.com

[app:main]
use = ...
set admin_email = bob@example.com
```

That is, by using `set` in front of the key.

Composite Applications

"Composite" applications are things that act like applications, but are made up of other applications. One example would be a URL mapper, where you mount applications at different URL paths. This might look like:

```
[composite:main]
use = egg:Paste#urlmap
/ = mainapp
/files = staticapp

[app:mainapp]
use = egg:MyApp

[app:staticapp]
use = egg:Paste#static
document_root = /path/to/docroot
```

The composite application "main" is just like any other application from the outside (you load it with `loadapp` for instance), but it has access to other applications defined in the configuration file.

Other Objects

In addition to sections with `app:`, you can define filters and servers in a configuration file, with `server:` and `filter:` prefixes. You load these with `loadserver` and `loadfilter`. The configuration works just the same; you just get back different kinds of objects.

Filter Composition

There are several ways to apply filters to applications. It mostly depends on how many filters, and in what order you want to apply them.

The first way is to use the `filter-with` setting, like:

```
[app:main]
use = egg:MyEgg
filter-with = printdebug

[filter:printdebug]
use = egg:Paste#printdebug
# and you could have another filter-with here, and so on...
```

Also, two special section types exist to apply filters to your applications: `[filter-app:...]` and `[pipeline:...]`. Both of these sections define applications, and so can be used wherever an application is needed.

`filter-app` defines a filter (just like you would in a `[filter:...]` section), and then a special key `next` which points to the application to apply the filter to.

`pipeline:` is used when you need apply a number of filters. It takes *one* configuration key `pipeline` (plus any global configuration overrides you want). `pipeline` is a list of filters ended by an application, like:

```
[pipeline:main]
pipeline = filter1 egg:FilterEgg#filter2 filter3 app

[filter:filter1]
...
```

Getting Configuration

If you want to get the configuration without creating the application, you can use the `appconfig(uri)` function, which is just like the `loadapp()` function except it returns the configuration that would be used, as a dictionary. Both global and local configuration is combined into a single dictionary, but you can look at just one or the other with the attributes `.local_conf` and `.global_conf`.

CHAPTER 11

egg: URIs

Python Eggs are a distribution and installation format produced by setuptools and distribute that adds metadata to a normal Python package (among other things).

You don't need to understand a whole lot about Eggs to use them. If you have a distutils `setup.py` script, just change:

```
from distutils.core import setup
```

to:

```
from setuptools import setup
```

Now when you install the package it will be installed as an egg.

The first important part about an Egg is that it has a *specification*. This is formed from the name of your distribution (the `name` keyword argument to `setup()`), and you can specify a specific version. So you can have an egg named `MyApp`, or `MyApp==0.1` to specify a specific version.

The second is *entry points*. These are references to Python objects in your packages that are named and have a specific protocol. "Protocol" here is just a way of saying that we will call them with certain arguments, and expect a specific return value. We'll talk more about the protocols *later*. The important part here is how we define entry points. You'll add an argument to `setup()` like:

```
setup(
    name='MyApp',
    ...
    entry_points={
        'paste.app_factory': [
            'main=myapp.mymodule:app_factory',
            'ob2=myapp.mymodule:ob_factory'],
        },
    )
```

This defines two applications named `main` and `ob2`. You can then refer to these by `egg:MyApp#main` (or just `egg:MyApp`, since `main` is the default) and `egg:MyApp#ob2`.

The values are instructions for importing the objects. `main` is located in the `myapp.mymodule` module, in an object named `app_factory`.

There's no way to add configuration to objects imported as Eggs.

CHAPTER 12

Defining Factories

This lets you point to factories (that obey the specific protocols we mentioned). But that's not much use unless you can create factories for your applications.

There's a few protocols: `paste.app_factory`, `paste.composite_factory`, `paste.filter_factory`, and lastly `paste.server_factory`. Each of these expects a callable (like a function, method, or class).

paste.app_factory

The application is the most common. You define one like:

```
def app_factory(global_config, **local_conf):
    return wsgi_app
```

The `global_config` is a dictionary, and local configuration is passed as keyword arguments. The function returns a WSGI application.

paste.composite_factory

Composites are just slightly more complex:

```
def composite_factory(loader, global_config, **local_conf):
    return wsgi_app
```

The `loader` argument is an object that has a couple interesting methods. `get_app(name_or_uri, global_conf=None)` return a WSGI application with the given name. `get_filter` and `get_server` work the same way.

A more interesting example might be a composite factory that does something. For instance, consider a "pipeline" application:

```
def pipeline_factory(loader, global_config, pipeline):
    # space-separated list of filter and app names:
    pipeline = pipeline.split()
    filters = [loader.get_filter(n) for n in pipeline[:-1]]
    app = loader.get_app(pipeline[-1])
    filters.reverse() # apply in reverse order!
    for filter in filters:
        app = filter(app)
    return app
```

Then we use it like:

```
[composite:main]
use = <pipeline_factory_uri>
pipeline = egg:Paste#printdebug session myapp

[filter:session]
use = egg:Paste#session
store = memory

[app:myapp]
use = egg:MyApp
```

paste.filter_factory

Filter factories are just like app factories (same signature), except they return filters. Filters are callables that take a WSGI application as the only argument, and return a "filtered" version of that application.

Here's an example of a filter that checks that the `REMOTE_USER` CGI variable is set, creating a really simple authentication filter:

```
def auth_filter_factory(global_conf, req_usernames):
    # space-separated list of usernames:
    req_usernames = req_usernames.split()
    def filter(app):
        return AuthFilter(app, req_usernames)
    return filter

class AuthFilter(object):
    def __init__(self, app, req_usernames):
        self.app = app
        self.req_usernames = req_usernames

    def __call__(self, environ, start_response):
        if environ.get('REMOTE_USER') in self.req_usernames:
            return self.app(environ, start_response)
        start_response(
            '403 Forbidden', [('Content-type', 'text/html')])
        return ['You are forbidden to view this resource']
```

paste.filter_app_factory

This is very similar to `paste.filter_factory`, except that it also takes a `wsgi_app` argument, and returns a WSGI application. So if you changed the above example to:

```
class AuthFilter(object):
    def __init__(self, app, global_conf, req_usernames):
        ....
```

Then `AuthFilter` would serve as a filter_app_factory (`req_usernames` is a required local configuration key in this case).

paste.server_factory

This takes the same signature as applications and filters, but returns a server.

A server is a callable that takes a single argument, a WSGI application. It then serves the application.

An example might look like:

```
def server_factory(global_conf, host, port):
    port = int(port)
    def serve(app):
        s = Server(app, host=host, port=port)
        s.serve_forever()
    return serve
```

The implementation of `Server` is left to the user.

paste.server_runner

Like `paste.server_factory`, except `wsgi_app` is passed as the first argument, and the server should run immediately.

CHAPTER 13

Outstanding Issues

- Should there be a "default" protocol for each type of object? Since there's currently only one protocol, it seems like it makes sense (in the future there could be multiple). Except that `paste.app_factory` and `paste.composite_factory` overlap considerably.
- ConfigParser's INI parsing is kind of annoying. I'd like it both more constrained and less constrained. Some parts are sloppy (like the way it interprets `[DEFAULT]`).
- `config:` URLs should be potentially relative to other locations, e.g., `config:$docroot/...`. Maybe using variables from `global_conf`?
- Should other variables have access to `global_conf`?
- Should objects be Python-syntax, instead of always strings? Lots of code isn't usable with Python strings without a thin wrapper to translate objects into their proper types.
- Some short-form for a filter/app, where the filter refers to the "next app". Maybe like:

```
[app-filter:app_name]
use = egg:...
next = next_app

[app:next_app]
...
```