plaster Documentation

Release 0.5

Michael Merickel

Jun 02, 2017

Contents

`plaster` is a loader interface around arbitrary config file formats. It exists to define a common API for applications to use when they wish to load configuration settings. The library itself does not aim to handle anything except a basic API that applications may use to find and load configuration settings. Any specific constraints should be implemented in a pluggable loader which can be registered via an entrypoint.

The library helps your application find an appropriate loader based on a *config uri* and a desired set of *loader protocol* identifiers.

Some possible `config_uri` formats:

- `development.ini`
- `development.ini#myapp`
- `development.ini?http_port=8080#main`
- `ini://development.conf`
- `pastedeploy+ini:///path/to/development.ini`
- `pastedeploy+ini://development.ini#foo`
- `egg:MyApp?debug=false#foo`

An example application that does not care what file format the settings are sourced from, as long as they are in a section named `my-settings`:

```python
import plaster
import sys

if __name__ == '__main__':
    config_uri = sys.argv[1]
    settings = plaster.get_settings(config_uri, 'my-settings')
```

This script can support any config format so long as the application (or the user) has installed the loader they expect to use. For example, `pip install plaster_pastedeploy`. The loader is then found by *`plaster.get_settings()`* based on the specific *config uri* provided. The application does not need to configure the loaders. They are discovered via pkg_resources entrypoints and registered for specific schemes.

CHAPTER 1

Protocols

`plaster` supports custom loader protocols which loaders may choose to implement to provide extra functionality over the basic *`plaster.ILoader`* interface. A *loader protocol* is intentionally very loosely defined but it basically boils down to a loader object that supports extra methods with agreed-upon signatures. Right now the only officially-supported protocol is `wsgi` which defines a loader that should implement the *`plaster.protocols.IWSGIProtocol`* interface.

CHAPTER 2

Known Loaders

- plaster_pastedeploy **officially supported**

  File types:

  - `file+ini`, `ini`, `pastedeploy+ini`
  - `egg`, `pastedeploy+egg`

  Protocols:

  - `wsgi` - *`plaster.protocols.IWSGIProtocol`*

CHAPTER 3

Installation

Stable release

To install plaster, run this command in your terminal:

```
$ pip install plaster
```

If you don't have pip installed, this Python installation guide can guide you through the process.

From sources

The sources for plaster can be downloaded from the Github repo.

```
$ git clone https://github.com/Pylons/plaster.git
```

Once you have a copy of the source, you can install it with:

```
$ pip install -e .
```

CHAPTER 4

Usage

Loading settings

A goal of `plaster` is to allow a configuration source to be used for multiple purposes. For example, an INI file is split into separate sections which provide settings for separate applications. This works because each application can parse the INI file easily and pull out only the section it cares about. In order to load settings, use the *`plaster.get_settings()`*.

The application may accept a path to a config file, allowing the user to specify the name of the section (`myapp`) to be loaded:

```
import plaster

config_uri = 'development.ini#myapp'
settings = plaster.get_settings(config_uri)
```

Alternatively, the application may depend on a specifically named section:

```
import plaster

config_uri = 'development.ini#myapp'
settings = plaster.get_settings(config_uri, section='thisapp')
```

Configuring logging

`plaster` requires a *loader* to provide a way to configure Python's stdlib logging module. In order to utilize this feature, simply call *`plaster.setup_logging()`* from your application.

```
import plaster

config_uri = 'redis://username@password:hostname/db?opt=val'
plaster.setup_logging(config_uri)
```

Finding a loader

At the heart of `plaster` is the `config_uri` format. This format is basically `<scheme>://<path>` with a few variations. The `scheme` is used to find an *`plaster.ILoaderFactory`*.

```
import plaster

config_uri = 'pastedeploy+ini://development.ini#myapp'
loader = plaster.get_loader(config_uri, protocols=['wsgi'])
settings = loader.get_settings()
```

A `config_uri` may be a file path or an **RFC 3986** URI. In the case of a file path, the file extension is used as the scheme. In either case the scheme and the protocols are the only items that `plaster` cares about with respect to finding an *`plaster.ILoaderFactory`*.

It's also possible to lookup the exact loader by prefixing the scheme with the name of the package containing the loader:

```
settings = plaster.get_settings('plaster_pastedeploy+ini://')
```

Writing your own loader

`plaster` finds loaders registered for the `plaster.loader_factory` entry point in your `setup.py`:

```
from setuptools import setup

setup(
    name='myapp',
    # ...
    entry_points={
        'plaster.loader_factory': [
            'dict = myapp:Loader',
        ],
    },
)
```

In this example the importable `myapp.Loader` class will be used as *`plaster.ILoaderFactory`* for creating *`plaster.ILoader`* objects. Each loader is passed a *`plaster.PlasterURL`* instance, the result of parsing the `config_uri` to determine the scheme and fragment.

If the loader should be found automatically via file extension then it should broadcast support for the special `file+<extension>` scheme. For example, to support `development.ini` instead of `myscheme://development.ini` the loader should be registered for the `file+ini` scheme.

```
import plaster

class Loader(plaster.ILoader):
    def __init__(self, uri):
        self.uri = uri

    def get_sections(self):
        return ['myapp', 'yourapp']

    def get_settings(self, section=None, defaults=None):
        # fallback to the fragment from config_uri if no section is given
        if not section:
            section = self.uri.fragment
        # if section is still none we could fallback to some
        # loader-specific default

        result = {}
        if defaults is not None:
            result.update(defaults)
        if section == 'myapp':
            result.update({'a': 1})
        elif section == 'yourapp':
            result.update({'b': 1})
        return result
```

This loader may then be used:

```
import plaster

settings = plaster.get_settings('dict://', section='myapp')
assert settings['a'] == 1

settings2 = plaster.get_settings('myapp+dict://', section='myapp')
assert settings == settings2
```

Supporting a custom protocol

By default, loaders are exposed via the `plaster.loader_factory` entry point. In order to register a loader that supports a custom protocol it should register itself on a `plaster.<protocol>_loader_factory` entry point.

A scheme **MUST** point to the same loader factory for every protocol, including the default (empty) protocol. If it does not then no compatible loader will be found if the end-user requests a loader satisfying both protocols.

CHAPTER 5

Acknowledgments

This API is heavily inspired by conversations, contributions, and design put forth in https://github.com/inklesspen/montague and https://metaclassical.com/announcing-montague-the-new-way-to-configure-python-applications/.

CHAPTER 6

More Information

plaster API

Application API

plaster.**get_settings**(*config_uri*, *section=None*, *defaults=None*)

Load the settings from a named section.

```
settings = plaster.get_settings(...)
print(settings['foo'])
```

Parameters

- **config_uri** – Anything that can be parsed by *plaster.parse_uri()*.
- **section** – The name of the section in the config file. If this is `None` then it is up to the loader to determine a sensible default usually derived from the fragment in the `path#name` syntax of the `config_uri`.
- **defaults** – A `dict` of default values used to populate the settings and support variable interpolation. Any values in `defaults` may be overridden by the loader prior to returning the final configuration dictionary.

Returns A `dict` of settings. This should return a dictionary object even if no data is available.

plaster.**setup_logging**(*config_uri*, *defaults=None*)

Execute the logging configuration defined in the config file.

This function should, at least, configure the Python standard logging module. However, it may also be used to configure any other logging subsystems that serve a similar purpose.

Parameters

- **config_uri** – Anything that can be parsed by *plaster.parse_uri()*.

- **defaults** – A `dict` of default values used to populate the settings and support variable interpolation. Any values in `defaults` may be overridden by the loader prior to returning the final configuration dictionary.

`plaster.`**`get_loader`**(*config_uri*, *protocols=None*)

Find a *`plaster.ILoader`* object capable of handling `config_uri`.

Parameters

- **config_uri** – Anything that can be parsed by *`plaster.parse_uri()`*.
- **protocols** – Zero or more *loader protocol* identifiers that the loader must implement to match the desired `config_uri`.

Returns A *`plaster.ILoader`* object.

Raises

- ***plaster.LoaderNotFound*** – If no loader could be found.
- ***plaster.MultipleLoadersFound*** – If multiple loaders match the requested criteria. If this happens, you can disambiguate the lookup by appending the package name to the scheme for the loader you wish to use. For example if `ini` is ambiguous then specify `ini+myapp` to use the ini loader from the `myapp` package.

`plaster.`**`find_loaders`**(*scheme*, *protocols=None*)

Find all loaders that match the requested scheme and protocols.

Parameters

- **scheme** – Any valid scheme. Examples would be something like `ini` or `ini+pastedeploy`.
- **protocols** – Zero or more *loader protocol* identifiers that the loader must implement. If `None` then only generic loaders will be returned.

Returns A list containing zero or more *`plaster.ILoaderInfo`* objects.

class `plaster.`**`ILoaderInfo`**

An info object describing a specific *`plaster.ILoader`*.

Variables

- **scheme** – The full scheme of the loader.
- **protocols** – Zero or more supported *loader protocol* identifiers.
- **factory** – The *`plaster.ILoaderFactory`*.

`load`(*config_uri*)

Create and return an *`plaster.ILoader`* instance.

Parameters **config_uri** – Anything that can be parsed by *`plaster.parse_uri()`*.

Loader API

class `plaster.`**`ILoader`**

An abstraction over an source of configuration settings.

It is required to implement `get_sections`, `get_settings` and `setup_logging`.

Optionally, it may also implement other *loader protocol* interfaces to provide extra functionality. For example, *`plaster.protocols.IWSGIProtocol`* which requires `get_wsgi_app`, and `get_wsgi_server` for loading WSGI configurations. Services that depend on such functionality should document the required functionality behind a particular *loader protocol* which custom loaders can implement.

Variables **uri** – The `plaster.PlasterURL` object used to find the `plaster.ILoaderFactory`.

get_sections()
Load the list of section names available.

get_settings(*section=None*, *defaults=None*)
Load the settings for the named `section`.

Parameters

- **section** – The name of the section in the config file. If this is `None` then it is up to the loader to determine a sensible default usually derived from the fragment in the `path#name` syntax of the `config_uri`.
- **defaults** – A `dict` of default values used to populate the settings and support variable interpolation. Any values in `defaults` may be overridden by the loader prior to returning the final configuration dictionary.

Returns A `dict` of settings. This should return a dictionary object even if the section is missing.

Raises **ValueError** – If a section name is missing and cannot be determined from the `config_uri`.

setup_logging(*defaults=None*)
Execute the logging configuration defined in the config file.

This function should, at least, configure the Python standard logging module. However, it may also be used to configure any other logging subsystems that serve a similar purpose.

Parameters **defaults** – A `dict` of default values used to populate the settings and support variable interpolation. Any values in `defaults` may be overridden by the loader prior to returning the final configuration dictionary.

class plaster.**ILoaderFactory**

__call__(*uri*)
A factory which accepts a `plaster.PlasterURL` and returns a `plaster.ILoader` object.

plaster.**parse_uri**(*config_uri*)
Parse the `config_uri` into a `plaster.PlasterURL` object.

`config_uri` can be a relative or absolute file path such as `development.ini` or `/path/to/development.ini`. The file must have an extension that can be handled by a `plaster.ILoader` registered with the system.

Alternatively, `config_uri` may be a **RFC 1738**-style string.

class plaster.**PlasterURL**(*scheme*, *path=''*, *options=None*, *fragment=''*)
Represents the components of a URL used to locate a `plaster.ILoader`.

Variables

- **scheme** – The name of the loader backend.
- **path** – The loader-specific path string. This is the entirety of the `config_uri` passed to `plaster.parse_uri()` without the scheme, fragment and options. If this value is falsey it is replaced with an empty string.

- **options** – A dictionary of options parsed from the query string as url-encoded key=value pairs.
- **fragment** – A loader-specific default section name. This parameter may be used by loaders in scenarios where they provide APIs that support a default name. For example, a loader that provides `get_wsgi_app` may use the fragment to determine the name of the section containing the WSGI app if none was explicitly defined. If this value is falsey it is replaced with an empty string.

Protocols

class `plaster.protocols.`**`IWSGIProtocol`**

`get_wsgi_app`(*name=None*, *defaults=None*)
Create a WSGI application object.

An example application object may be:

```
def app(environ, start_response):
    start_response(b'200 OK', [(b'Content-Type', b'text/plain')])
    yield [b'hello world\n']
```

Parameters

- **name** – The name of the application referenced in the config. If `None` then it should default to the `plaster.PlasterURL.fragment`, if available.
- **defaults** – A `dict` of default values used to populate the settings and support variable interpolation. Any values in `defaults` may be overridden by the loader prior to returning the final configuration dictionary.

Raises **LookupError** – If a WSGI application cannot be found by the specified name.

`get_wsgi_app_settings`(*name=None*, *defaults=None*)
Create a WSGI application object.

An example application object may be:

```
def app(environ, start_response):
    start_response(b'200 OK', [(b'Content-Type', b'text/plain')])
    yield [b'hello world\n']
```

Parameters

- **name** – The name of the application referenced in the config. If `None` then it should default to the `plaster.PlasterURL.fragment`, if available.
- **defaults** – A `dict` of default values used to populate the settings and support variable interpolation. Any values in `defaults` may be overridden by the loader prior to returning the final configuration dictionary.

Raises **LookupError** – If a WSGI application cannot be found by the specified name.

`get_wsgi_filter`(*name=None*, *defaults=None*)
Create a composable WSGI middleware object.

An example middleware filter may be:

```
class Filter(object):
    def __init__(self, app):
        self.app = app

    def __call__(self, environ, start_response):
        return self.app(environ, start_response)
```

Parameters

- **name** – The name of the application referenced in the config. If None then it should default to the plaster.PlasterURL.fragment, if available.
- **defaults** – A dict of default values used to populate the settings and support variable interpolation. Any values in defaults may be overridden by the loader prior to returning the final configuration dictionary.

Raises LookupError – If a WSGI filter cannot be found by the specified name.

get_wsgi_server(*name=None*, *defaults=None*)
Create a WSGI server runner.

An example server runner may be:

```
def runner(app):
    from wsgiref.simple_server import make_server
    server = make_server('0.0.0.0', 8080, app)
    server.serve_forever()
```

Parameters

- **name** – The name of the application referenced in the config. If None then it should default to the plaster.PlasterURL.fragment, if available.
- **defaults** – A dict of default values used to populate the settings and support variable interpolation. Any values in defaults may be overridden by the loader prior to returning the final configuration dictionary.

Raises LookupError – If a WSGI server cannot be found by the specified name.

Exceptions

exception plaster.**PlasterError**
A base exception for any error generated by plaster.

exception plaster.**InvalidURI**(*uri*, *message=None*)
Raised by *plaster.parse_uri()* when failing to parse a config_uri.

Variables uri – The user-supplied config_uri string.

exception plaster.**LoaderNotFound**(*scheme*, *protocols=None*, *message=None*)
Raised by *plaster.get_loader()* when no loaders match the requested scheme.

Variables

- **scheme** – The scheme being matched.
- **protocols** – Zero or more *loader protocol* identifiers that were requested when finding a loader.

exception `plaster.`**`MultipleLoadersFound`**(*scheme*, *loaders*, *protocols=None*, *message=None*)

Raised by *`plaster.get_loader()`* when more than one loader matches the requested `scheme`.

Variables

- **`scheme`** – The scheme being matched.
- **`protocols`** – Zero or more *loader protocol* identifiers that were requested when finding a loader.
- **`loaders`** – A list of *`plaster.ILoaderInfo`* objects.

Glossary

config uri In most cases this is simply an absolute or relative path to a config file on the system. However, it can also be a **RFC 1738**-style string pointing at a remote service or a specific parser without relying on the file extension. For example, `my-ini://foo.ini` may point to a loader named `my-ini` that can parse the relative `foo.ini` file.

loader An object conforming to the *`plaster.ILoader`* interface. A loader is responsible for locating and parsing the underlying configuration format for the given *config uri*.

loader protocol A *loader* may implement zero or more custom named protocols. An example would be the `wsgi` protocol which requires that a loader implement certain methods like `wsgi_app = get_wsgi_app(name=None, defaults=None)`.

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/Pylons/plaster/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

plaster could always use more documentation, whether as part of the official plaster 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/Pylons/plaster/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 *plaster* for local development.

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

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

3. Install your local copy into a virtualenv:

   ```
   $ python3 -m venv env
   $ env/bin/pip install -e .[docs,testing]
   $ env/bin/pip install 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 flake8 and the tests, including testing other Python versions with tox:

   ```
   $ env/bin/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 with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.7, 3.4, 3.5, 3.6, and for PyPy. Check https://travis-ci.org/Pylons/plaster/pull_requests and make sure that the tests pass for all supported Python versions.

Tips

To run a subset of tests:

```
$ env/bin/py.test tests.test_plaster
```

Changes

0.5 (2017-06-02)

- When a scheme is not supplied, `plaster.parse_uri` will now autogenerate a scheme from the file extension with the format `file+<ext>` instead of simply `<ext>` (for example, `file+ini` instead of `ini`). See https://github.com/Pylons/plaster/pull/16
- Absolute lookups are now pulled from the start of the scheme instead of the end. This means that if you want to explicitly define the package that the loader is pulled from, use `package+scheme` instead of `scheme+package`. See https://github.com/Pylons/plaster/pull/16

0.4 (2017-03-30)

- Removed the `plaster.NoSectionError` exception. It's expected that individual loaders should return an empty dictionary of settings in the case that a section cannot be found. See https://github.com/Pylons/plaster/pull/12
- Expect the `wsgi` protocol to raise `LookupError` exceptions when a named wsgi component cannot be found. See https://github.com/Pylons/plaster/pull/12

0.3 (2017-03-27)

- Lookup now works differently. First "foo+bar" looks for an installed project distribution named "bar" with a loader named "foo". If this fails then it looks for any loader named "foo+bar".
- Rename the loader entry point to `plaster.loader_factory`.
- Add the concept of protocols to `plaster.get_loader` and `plaster.find_loaders`.
- `plaster.find_loaders` now works on just schemes and protocols instead of full `PlasterURL` objects and implements the lookup algorithm for finding loader factories.
- Change the `ILoaderInfo` interface to avoid being coupled to a particular uri. `ILoaderInfo.load` now takes a `config_uri` parameter.
- Add a `options` dictionary to `PlasterURL` containing any arguments decoded from the query string. Loaders may use these for whatever they wish but one good option is default values in a config file.
- Define the `IWSGIProtocol` interface which addons can use to implement a loader that can return full wsgi apps, servers and filters.

- The scheme is now case-insensitive.

0.2 (2016-06-15)

- Allow `config_uri` syntax `scheme:path` alongside `scheme://path`. See https://github.com/Pylons/plaster/issues/3
- Improve errors to show the user-supplied values in the error message. See https://github.com/Pylons/plaster/pull/4
- Add `plaster.find_loaders` which can be used by people who need a way to recover when ambiguous loaders are discovered via `plaster.get_loader`. See https://github.com/Pylons/plaster/pull/5
- Rename `plaster.Loader` to `plaster.ILoader` to signify its purpose as an interface with no actual implementation. See https://github.com/Pylons/plaster/pull/5
- Introduce `plaster.ILoaderFactory` to document what the entry point targets are expected to implement. See https://github.com/Pylons/plaster/pull/5

0.1 (2016-06-12)

- Initial release.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index