Confire Documentation

Release 0.2.0

Benjamin Bengfort

December 10, 2016

Contents

Confire is a simple but powerful configuration scheme that builds on the configuration parsers of Scapy, elasticsearch, Django and others. The basic scheme is to have a configuration search path that looks for YAML files in standard locations. The search path is hierarchical (meaning that system configurations are overloaded by user configurations, etc). These YAML files are then added to a default, class-based configuration management scheme that allows for easy development.

CHAPTER 1

Features

- Configuration files in YAML
- Hierarchical configuration search
- Class based application defaults
- Settings pulled in from the environment

CHAPTER 2

Setup

The easiest and usual way to install confire is to use pip:

```
pip install confire
```

To install the package from source, download the latests package tarball, unzip in a temporary directory and run the following command:

```
python setup.py install
```

As always, I highly recommend the use of a virtual environment to better manage the software dependencies for your particular code base.

Example Usage

Create a file called "myapp.yaml" and place it in one of the following places:

- `/etc/myapp.yaml`
- `$HOME/.myapp.yaml`
- `conf/myapp.yaml`

Create some configuration values inside the file like so:

```yaml
## Set application environment
debug:   True
testing: False

## A simple database configuration
database:
    name: mydb
    host: localhost
    port: 5432
    user: postgres
```

In your code, create a file called "config.py" and add the following:

```python
import os
from confire import Configuration
from confire import environ_setting

class DatabaseConfiguration(Configuration):

    host = "localhost"
    port = 5432
    name = "mydb"
    user = "postgres"
    password = environ_setting("DATABASE_PASSWORD", required=False)

class MyAppConfiguration(Configuration):

    CONF_PATHS = [
        '/etc/myapp.yaml',
        os.path.expanduser('~/.myapp.yaml'),
        os.path.abspath('conf/myapp.yaml')
    ]

    debug    = False
    testing  = True
    database = DatabaseConfiguration()

settings = MyAppConfiguration.load()
```

Now, everywhere in your code that you would like to access these settings values, simply use as follows:

```
from config import settings

debug = settings.get('DEBUG') or settings['DEBUG']
```

Voila! A complete configuration system for your application!

CHAPTER 4

Next Topics

Here are a list of topics for more detail about how to use confire in your Python applications.

4.1 Configuring Apps with Confire

Let's say that you've just started a new Python project - you know that this project is going to need access to a Database, possibly require an API key and API Secret, and it will definitely need some sort of debug mode so that developers can figure out what's going on in production. These types of variables shouldn't be hardcoded into your application, you'll want some kind of configuration management system in your app.

So what are your options? Python has a native configuration parser that handles .ini files similar to what you'd see on Windows machines - it's called `configparser`, and while it works well (and even has support for JSON files) - it is extremely basic. As a developer, you not only have to deal with the .ini syntax, but you also have to look for the file and load it into the parser. While the parser does handle type conversion, it has no quick ability to add reasonable defaults. Basically, your configuration ends up being defined by the .ini file - and this is not good, especially if your users forget to change a particular value or leave one out all together!

If you look at Django, they have their settings in a Python file, and the settings are Python code. This is great because now you can use any Python type as a setting. There are even reasonable defaults and some fancy import logic helps get the settings where they need to be. The problem is that you have to import that file, it has to be on your python path - so no storage of a settings file in /etc or any other reasonable place. It also means that *developers* have to configure Django- not just users of the app.

So we want the following things in our configuration:

1. Reasonable lookup locations for config files
2. Configuration type parsing from a text file
3. An API that supports reasonable defaults and in-code usage
4. A text based configuration that is for users not developers

This is where confire comes in. Confire uses YAML as the configuration language of choice. This is a markup format that has rich types like JSON, but is also very readable. Applications like Elasticsearch, Ruby on Rails, Travis-CI and others make use of YAML, so it's probably already familiar to you.

Confire has a hierarchical lookup system that means it looks in the system configuration (`/etc` in *nix systems), then in a user specific place, then in a local directory. At each level, the configuration overrides the defaults from the other levels. Configurations are then supplied to the developer in a friendly, Django-like way.

4.1.1 Project Setup

In your projects folder, create your app folder, let's call it "myapp". Then create the Python project skeleton as you would normally do, but also include a configuration directory, "conf".

```
$ mkdir myapp
$ cd myapp
$ mkdirs bin tests conf docs fixtures myapp
$ touch tests/__init__.py
$ touch myapp/__init__.py
$ touch setup.py
$ cd docs
$ sphinx-quickstart
...
$ cd ..
```

Hopefull this is very familiar to those who develop on Linux or Mac and set up Python projects regularly. Now, assuming you're using Git as well as virtualenv and virtualenv wrapper - let's get our repository and env going:

```
$ git init
$ mkvirtualenv -a $(pwd) myapp
(myapp)$ pip install confire
(myapp)$ pip install nose
(myapp)$ pip freeze > requirements.txt
```

Perfect! You're now ready to get developing your Python app. Let's start by getting our configuration going. Create a file called `myapp.yaml` in the `conf` directory. It may be helpful to add this file to your `.gitignore` so that you don't accidentally commit a private variable publically. Also create a file called `config.py` in your `myapp` module.

Inside the `myapp.yaml` file place the following, very simple code.

```
debug: false
testing: false
```

And then inside your `config.py`, place the following code.

```
import os
from confire import Configuration

class MyAppConfiguration(Configuration):

    CONF_PATHS = [
        "/etc/myapp.yaml",                          # System configuration
        os.path.expandvars("$HOME/.myapp.yaml"),    # User specific configuration
        os.path.abspath("conf/myapp.yaml"),         # Development configuration
    ]

    debug   = True
    testing = True

## Load settings immediately for import
settings = MyAppConfiguration.load()

if __name__ == "__main__":
    print settings
```

That's it, you now have a complete configuration system for your app! Let's walk through this code. Confire provides a class-based configuration API, meaning that you simply create configuration classes and then defeine your defaults on them at the class level (kind of like you might use Django class-based views). Configuration classes must all extend the `confire.Configuration` base class.

Note: All configurations should be lowercase properties! Configurations are case insensitive, but to achieve this, the `__getitem__` method lowercases all accessors!

The `CONF_PATHS` class variable tells the configuration where to look for YAML files to load. In this case, we specify three lookups that happen in the order they're specified - first the system, then the user directory, then the local directory for development. You'll notice that if the config file is missing, no exceptions are raised.

4.1.2 Using Configurations in Code

The loaded settings immediately for import means that elsewhere in your code, all you have to do is use the following to get access to your config:

```
from myapp.config import settings

if settings.get("DEBUG"):
    ...
else:
    ...
```

Because your API has already specified reasonable defaults, you don't have to worry about configurations being missing or unavailable!

A couple notes on using the settings in your code:

1. The settings *are not* case sensitive, DEBUG is the same as debug. However, all properties should be stored as lowercase in the configuration subclass.
2. You can access settings like so: `settings["mysetting"]`, however this will raise an exception if the setting is not available (something that really shouldn't happen).
3. You can also access the settings through the get method: `settings.get("mysetting", "foo")`, which will not raise an exception on a missing setting, but instead return the supplied default or `None`.
4. You can also access the settings using a dot accessor method: `settings.mysetting`, which fetches the properties off the class.
5. Settings can be modified at runtime, but this is not recommended.

As you continue to develop, you can add settings to your `config.py` as well as your `myapp.yaml`, your app development is now much smoother!

4.2 Environment Variables

Sometimes you don't want your configurations to reside inside of a YAML file, saved on disk, usually when you have a secret key or a database password. Other times you don't have access to your server's disk, but can add ENVIRONMENTAL VARIABLES as with a hosting service like Heroku.

Confire makes it easy to specify variables that you expect to be in the environment, using the `environ_setting` function, which you can import from the main module.

```
from confire import Configuration, environ_setting

class MyConfiguration(Configuration):

    supersecret = environ_setting("SUPER_SECRET", None, required=True)
```

The function expects as a first argument, the name of the environment variable, usually an all caps, underscore separated name. You can also give a default value (in case no variable exists in the environment) as the second argument.

When the environment is initialized (not loaded) it will immediately look in the environment for the setting and store it as the default. Any settings that are in the YAML search paths will override the environment variable, so make sure that you leave ENV_VARS out of the YAML configs!

The behavior of the function depends on how it's called, in terms of using the default and fetching from the environment:

1. If it is required and the default is None, raise ImproperlyConfigured
2. If it is requried and a default exists, return default
3. If it is not required and default is None, return None
4. If it is not required and default exists, return default

Environmental variables are usually required, hence the exception.

Note also that you can use confire exceptions and warnings in your own code, by importing the `ImproperlyConfigured` and `MissingConfiguration` exception and warning.

4.3 Nested Configurations

Configurations are nestable in order to ensure that developers can create easily modular configurations, for example database configuations for a staging and production database or per-app settings. Nested configurations will also be loaded from a single YAML file that expects a similar nesting structure, and the configurations are loaded in a depth-first manner.

To create a nested configuration, you need a main configuration object that supports the top-level configuration. For each nested configuration, you simply create new `Configuration` subclasses and then add them as settings to main configuration class.

Here is the example for two different databases:

```
import confire

class DatabaseConfiguration(confire.Configuration):

    name   = None
    host   = "localhost"
    port   = 5432
    user   = None
    pass   = None

class MainConfiguration(confire.Configuration):

    staging    = DatabaseConfiguration()
    production = DatabaseConfiguration()

settings = MainConfiguration.load()
```

In your YAML file, you can configure each database configuration for its specific environment:

```
staging:
    name: "myapp-staging"
    host: "localhost"
    port: 5432
    user: "test-user"
    pass: "password"
production:
    name: "myapp-production"
    host: "54.21.35.141"
    port: 5432
    user: "user"
    pass: "password"
```

Access to the configuration is as follows:

```
from myapp.config import settings

print settings.staging.host
print settings.production.host
```

Configurations can be nested to any depth, but it is recommended to keep them fairly shallow, to avoid deep accessor chains.

4.4 Class Documentation

This page documents the contents of the library.

4.4.1 Confire Module

A simple app configuration scheme using YAML and class based defaults.

`confire.`**`get_version`**`(`*short=False*`)`
    Prints the version.

4.4.2 Config Module

Confire class for specifying Confire specific optional items via a YAML configuration file format. The main configuration class provides utilities for loading the configuration from disk and iterating across all the settings. Subclasses of the Configuration specify defaults that can be updated via the configuration files.

General usage:

    from confire.conf import settings mysetting = settings.get('mysetting', default)

You can also get settings via a dictionary like access:

    mysetting = settings['mysetting']

However, this will raise an exception if the setting is not found.

Note: Keys are CASE insensitive

Note: Settings can be modified directly by settings.mysetting = newsetting however, this is not recommended, and settings should be fetched via the dictionary-like access.

class `confire.config.`**`Configuration`**
    Base configuration class specifies how configurations should be handled and provides helper methods for iterating through options and configuring the base class.

    Subclasses should provide defaults for the various configurations as directly set class level properties. Note, however, that ANY directive set in a configuration file (whether or not it has a default) will be added to the configuration.

Example:

class MyConfig(Configuration):

mysetting = True logpath = "/var/log/myapp.log" appname = "MyApp"

The configuration is then loaded via the classmethod *load*:

settings = MyConfig.load()

Access to properties is done two ways:

settings['mysetting'] settings.get('mysetting', True)

Note: None settings are not allowed!

`configure` (*conf={}*)
Allows updating of the configuration via a dictionary of configuration terms or a configuration object. Generally speaking, this method is utilized to configure the object from a JSON or YAML parsing.

`get` (*key*, *default=None*)
Fetches a key from the configuration without raising a KeyError exception if the key doesn't exist in the config, instead it returns the default (None).

classmethod `load` (*klass*)
Insantiates the configuration by attempting to load the configuration from YAML files specified by the CONF_PATH module variable. This should be the main entry point for configuration.

`options` ()
Returns an iterable of sorted option names in order to loop through all the configuration directives specified in the class.

`confire.config.`**`environ_setting`** (*name*, *default=None*, *required=True*)
Fetch setting from the environment. The bahavior of the setting if it is not in environment is as follows:

1. If it is required and the default is None, raise Exception
2. If it is requried and a default exists, return default
3. If it is not required and default is None, return None
4. If it is not required and default exists, return default

4.4.3 Exceptions

Exceptions hierarchy for Confire

exception `confire.exceptions.`**`ConfigurationMissing`**
Warn the user that an optional configuration is missing.

exception `confire.exceptions.`**`ConfireException`**
Base class for configuration exceptions.

exception `confire.exceptions.`**`ConfireWarning`**
Base class for configuration warnings.

exception `confire.exceptions.`**`ImproperlyConfigured`**
The user did not properly set a configuration value.

4.4.4 Examples

This file is an example file that you might put into your code base to have a configuration library at your fingertips!

class `confire.example.`**`DatabaseConfiguration`**

This object contains the default connections to a Postgres Database.

class `confire.example.`**`ExampleConfiguration`**

This object contains an example configuration.

debug: allow debug checking testing: are we in testing mode?

4.5 Changelog

The release versions that are sent to the Python package index are also tagged in Github. You can see the tags through the Github web application and download the tarball of the version you'd like. Additionally PyPI will host the various releases of confire.

The versioning uses a three part version system, "a.b.c" - "a" represents a major release that may not be backwards compatible. "b" is incremented on minor releases that may contain extra features, but are backwards compatible. "c" releases are bugfixes or other micro changes that developers should feel free to immediately update to.

4.5.1 Contributors

I'd like to personally thank the following people for contributing to confire and making it a success!

- @tyrannosaurus
- @murphsp1
- @keshavmagge
- @ojedatony1616

4.5.2 Versions

The following lists the various versions of confire and important details about them.

v0.2.0

- **tag**: v0.2.0
- **deployment**: July 31, 2014
- **commit**: (latest)

This release added some new features including support for environmental variables as settings defaults, ConfigurationMissing Warnings and ImproperlyConfigured errors that you can raise in your own code to warn developers about the state of configuration.

This release also greatly increased the amount of available documentation for Confire.

v0.1.1

- **tag**: v0.1.1
- **deployment**: July 24, 2014
- **commit**: bdc0488

Added Python 3.3 support thanks to @tyrannosaurus who contributed to the changes that would ensure this support for the future. I also added Python 3.3 travis testing and some other minor changes.

v0.1.0

- **tag**: v0.1.0
- **deployment**: July 20, 2014
- **comit**: 213aa5e

Initial deployment of the confire library.

CHAPTER 5

About

There are many configuration packages available on PyPI - it seems that everyone has a different way of doing it. However, this is my prefered way, and I found that after I copy and pasted this code into more than 3 projects that it was time to add it as a dependency via PyPI. The configuration builds on what I've learned/done in configuring Scapy, elasticsearch, and Django - and builds on these principles:

1. Configuration *should not* be Python (sorry Django). It's too easy to screw stuff up, and anyway, you don't want to deal with importing a settings file from `/etc`!
2. Configuration should be on a per-system basis. This means that there should be an `/etc/app.yaml` configuration file as well as a `$HOME/.app.yaml` configuration file that overwrites the system defaults for a particular user. For development purposes there should also be a `$(pwd)/app.yaml` file so that you don't have to sprinkle things throughout the system if not needed.
3. Developers should be able to have reasonable defaults already written in code if no YAML file has been provided. These defaults should be added in an API like way that is class based and modularized.
4. Accessing settings from the code should be easy.

So there you have it, with these things in mind I wrote confire and I hope you enjoy it!

5.1 Contributing

Confire is open source, and I would be happy to have you contribute! You can contribute in the following ways:

1. Create a pull request in Github: https://github.com/bbengfort/confire
2. Add issues or bugs on the bugtracker: https://github.com/bbengfort/confire/issues
3. Checkout the current dev board on waffle.io: https://waffle.io/bbengfort/confire

You can contact me on Twitter if needed: @bbengfort

5.2 Name Origin

I like cooking, and the thought of preparation in French culinary language appealed to me. The way I got here was to simply change the "g" in config to a "t". A definition lookup and boom, a name!

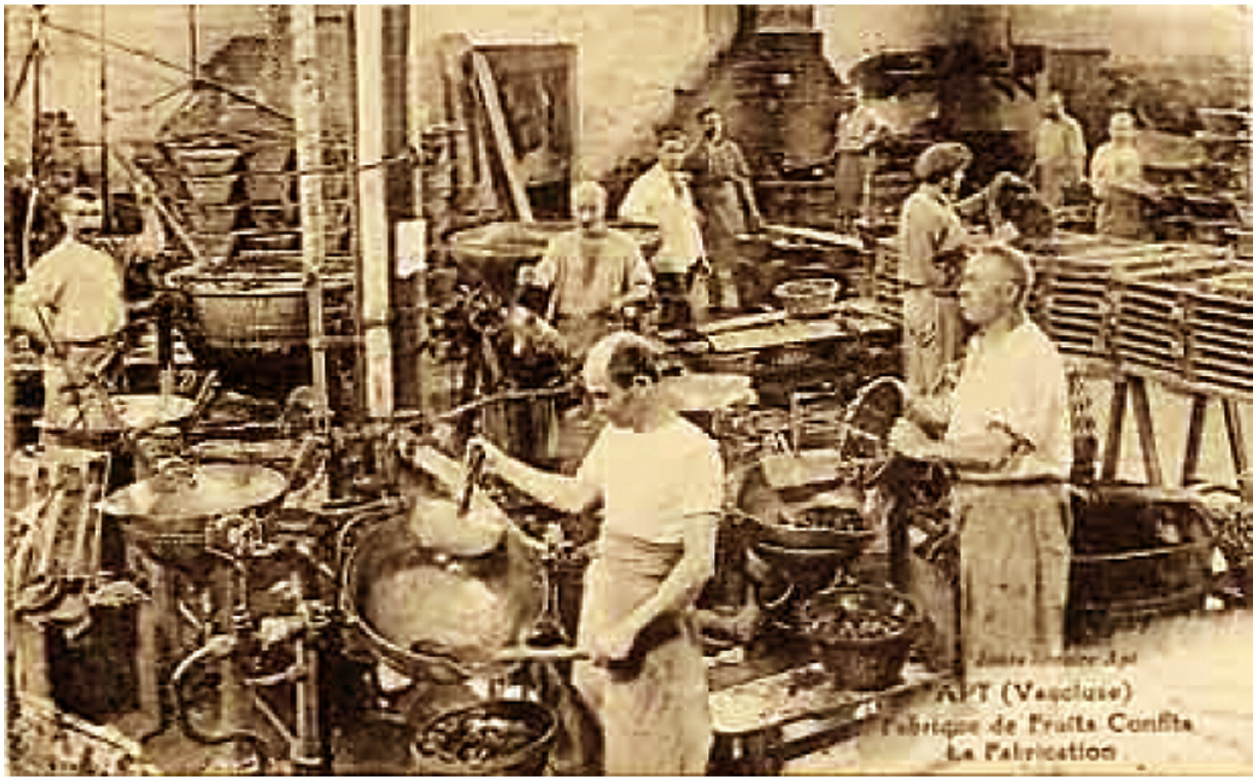

Fig. 5.1: French Chefs preparing Confit

5.3 Indices and tables

- genindex
- modindex
- search

Python Module Index

Index