# YamJam Documentation

***Release 0.1.5***

**Jeff Hinrichs**

April 28, 2014

# Contents

*keeping private data out of source control and applying DRY principles for resource information since 2009*

CHAPTER 1

# Overview

is a multi-project, shared, yaml based configuration system. It is also a mechanism to keep secret/private data from leaking out to source control systems (i.e. git, bitbucket, svn, et al) by factoring out sensitive data from your commits.

Tested on Python 2.7, 3.2, 3.3, 3.4

CHAPTER 2

# Installation

```
pip install yamjam
mkdir ~/.yamjam
touch ~/.yamjam/config.yaml
chmod -R go-rwx ~/.yamjam
```

Now open ~/.yamjam/config.yaml with your editor of choice and start factoring out your sensitive information. *Organizing your config.yaml*

CHAPTER 3

# What Next?

- **License**: BSD
- **Documentation**: You are here
- **Project**: View
- **Contributing**: Do

We work so well with Django, you'd think we should spell our name YamDjam

Contents:

## 3.1 Organizing your config.yaml

### 3.1.1 Location of config.yaml

The **base** yamjam config file is located *~/.yamjam/config.yaml*

- on windows:

  c:\documents and settings\[username]\.yamjam\config.yaml

- on unix:

  ~/.yamjam/config.yaml

Then yamjam looks for possible overrides/additions in the current working directory of the *APP* for a file named config.yaml (if it exists) This would be the **project** config.yaml

You can override the config file(s) when you call the function by specifying a different one explicitly. i.e.

```
cfg = yamjam('/path/to/file/file')
```

By default YamJam looks for two config.yaml files, the **base** and then a **project** specific one. You may specify one or more files. Each file path listed should be separated by semi-colons(;) NOTE: if you specify files, YamJam then only looks for them and not the default base and project files.

```
cfg = yamjam('/path/to/file/file1;/other/path/file2')
```

### 3.1.2 Base config only

The vast majority of use cases can be accomplished with just a base config located at *~/.yamjam/config.yaml* be hesitant to implement a project level config.yaml until you hit an edge case that requires it. This is the primary way of using

YamJam. The project level config.yaml and user specified locations cause as many problems as the solve and or only applicable to a small set of edge cases. You will be best served by not straying from the KISS principle.

### 3.1.3 Structure of config.yaml

config.yaml is a straight up yaml file. See http://pyyaml.org/ and http://www.yaml.org/ for advanced usage. However, if you write python you know what you need to write decent yaml. White space is relevant and that is a good thing! What makes Python a joy to read is the same for yaml.

### 3.1.4 Example

if you edit your ~/.yamjam/config.yaml file to include

```
myproject:
    key1: value
    key2: 42
```

It is returned from yamjam() as a dictionary.

```
{'myproject':
    {'key1': 'value',
     'key2': 42
    }
}
```

to get the value of key2, under myproject

```
>>> yamjam()['myproject']['key2']
42
```

Ok, now you know how to set up a dictionary in yaml, here is how you set up a list for a value.

edit config.yaml

```
myproject:
    key1: value
    key2: [42, 21, 7]
```

Now the value of key2 will return a list with 3 integers.

```
>>> yamjam()['myproject']['key2']
[42, 21, 7]
```

Now you are truly dangerous. You can create a dictionary of dictionaries that can contain more dictionaries, lists, integers, floats, strings, etc.

In fact yaml and pyyaml can do some wonderfully advanced stuff. But, always tend to the simplest form that will get the job done, think DRY KISS.

### 3.1.5 Logical Alignment

Since your config.yaml can store settings for more than one project and it almost always will, the standard use case is to create a section named for the project that uses it. i.e. If you have a Django project named ‘myproject’ then create a section in your config.yaml named myproject.

```
myproject:
    django-secret-key: my-secret-key
```

Ok, now you will need database connection information. So let's add it.

```
myproject:
    django-secret-key: my-secret-key
    database:
        engine: django.db.backends.postgresql_psycopg2
        name: mydatabase
        user: mydatabaseuser
        password: mypassword
        host: 127.0.0.1
        port: 5432
```

Now let's create config settings for our next-project

```
myproject:
    django-secret-key: my-secret-key
    database:
        engine: django.db.backends.postgresql_psycopg2
        name: mydatabase
        user: mydatabaseuser
        password: mypassword
        host: 127.0.0.1
        port: 5432

next-project:
    django-secret-key: next-project-secret-key
    database:
        engine: django.db.backends.postgresql_psycopg2
        name: mynextdatabase
        user: mynextdatabaseuser
        password: mynextpassword
        host: 127.0.0.1
        port: 5432
```

Ok, let's access the django-secret-key for next-project

```
>>> yamjam()['next-project']['django-secret-key']
next-project-secret-key
```

So, now you can start factoring out your sensitive data from your app. One more thing, in case it is not obvious, you can reduce your typing by caching just the config information you need to access.

```
>>> cfg = yamjam()['next-project']
>>> cfg['django-secret-key']
next-project-secret-key
```

That will keep your code looking clean and also, you just cached the results of the yamjam() call. Further accesses to cfg now happen at break-neck python speed.

Each time your call yamjam() it re-reads the config files. Sometimes this is desired, sometimes not but as a programmer you have complete and simple control over how things happen.

### 3.1.6 Merging Config Data

When more than one config.yaml is processed, base and project are merged using the following 2 rules.

If the value being merged is not a dictionary then the base value is replaced by the project.

> {'key1': val1} merged with {'key1': val2} results in {'key1': val2}

If the value being merged is a dictionary, then it is merged.

> {'key1': {'foo': 2}} merged with {'key1': {'bar': 3}} results in {'key1': {'foo': 2, 'bar': 3}}
>
> and
>
> {'key1': {'foo': 2}} merged with {} results in {'key1': {'foo': 2}}

This way the project config only specified overrides and additions to the base config. See the tests for test_merge.py for more examples. But the net result of the merge should not surprise you and allow for you to supply the minimal amount of project config data needed to modify the base config.

## 3.2 Django Usage

### 3.2.1 Django Scenario - Sensitive Data

settings.py contains a number of configuration settings that should be kept secret/private and not be included in commits to source code control. So you either don't check in your settings.py file (which creates its own set of problems) or your roll your own system. YamJam takes care of the tedious for you.

#### Example

*settings.py*

```
from YamJam import yamjam
...
DJANGO_SECRET_KEY = yamjam()['myproject']['django_secret_key']
...
```

*~/.yamjam/config.yaml*

```
myproject:
    django_secret_key: my-secret-key-value
```

### 3.2.2 Django Scenario - Differing Settings for Development/ Staging/ and Production

Your dev system uses a local data store for on the go development and to keep any migrations or bad data out of your staging and/or production environments. This means you need a different settings.py for each environment. Ok, there is a problem. Wouldn't you prefer one settings.py file with the specialized settings handled without writing lots of if/else statements? Enter YamJam

#### Example

*settings.py*

```
from YamJam import yamjam
...
dbcfg = yamjam()['myproject']['database']
```

```
DATABASES = {
    'default': {
        'ENGINE':dbcfg['engine'],
        'NAME': dbcfg['name'],
        'USER': dbcfg['user'],
        'PASSWORD': dbcfg['password'],
        'HOST': dbcfg['host'],
        'PORT': dbcfg['port'],
    }
}
```

*~/.yamjam/config.yaml - on dev machine*

```
myproject:
    django_secret_key: my-secret-key-value
    database:
      engine: django.db.backends.postgresql_psycopg2
      name: mydatabase
      user: mydatabaseuser
      password: mypassword
      host: 127.0.0.1
      port: 5432
```

*~/.yamjam/config.yaml - on production machine*

```
myproject:
    django_secret_key: my-secret-key-value
    database:
      engine: django.db.backends.postgresql_psycopg2
      name: mydatabase
      user: mydatabaseuser
      password: mypassword
      host: 10.0.0.24
      port: 5432
```

Notice how you can use one settings.py, with sensitive information masked. Combined with different ~/.yamjam/config.yaml on each machine makes a drop dead simple, deploy by version control system.

#### Huzzah!

We keep the cold stuff cold and the hot stuff hot – Or should I say YamJam keeps the versioned stuff versioned and the private stuff private.

## 3.3 DRY Scenario

### 3.3.1 Multiple apps that access common resources

Do you ever write apps that integrate with other resources? Do you write additional apps that also integrate or report on those resources? If you find yourself not following the Don't Repeat Yourself (DRY) conventions for resource information then YamJam is your friend.

- App1 access the corporate accounting db and the mail server and your no-sql data to automate reporting to vendors or customers.
- App2 access the corporate accounting db and your no-sql data and your fax system to generate POs.

#### Change Happens

Your accounting db just moved to new hardware and now has a new address and user credentials. What do you do? Maybe each app has a settings/config file and the info is not hard coded but you still have to update the resource information in multiple places. Oh yeah, you need development set ups for those apps too.

#### Handle It.

If you use YamJam you'll have a single point to update resource information, dev and production systems settings. Win, Win, Win! YamJam is not just for Django it is for any app that needs to be able to keep private information private and create a DRY environment for your shared resource configurations.

### 3.3.2 Advanced DRY usage

The following is not specific to YamJam as it is in the YAML specs.

YAML allows for Alias indicators which allow for the following example:

```
bill-to: &id001
    given  : Monty
    family : Python
ship-to: *id001
```

which would return the following:

```
{'bill-to': {'given': 'Monty', 'family': 'Python'}, 'ship-to': {'given': 'Monty', 'family': 'Python'
```

This is helpful when you wish to refactor a key for a setting but you have other apps still relying on the old key. Until you can refactor those apps to use the new key, you can use an Alias to keep your information DRY

## 3.4 Frequently asked questions

### 3.4.1 Why yaml for persistence?

All config dialects have their warts, some have more than others. However, YAML, in our opinion, is the most pythonic of the bunch. White space is relevant in yaml just like Python. Also, the visual look of yaml for dictionaries and lists is very congruous to Python.

```
key: value
adict:
    key: value
    key2: value2
alist: [1, 2, 3, 4]
```

### 3.4.2 How do I write to my config?

The short answer, is with your editor. YamJam is designed for change infrequently, read frequently scenarios. Your Django apps don't write to settings.py, we don't write to disk either.

### 3.4.3 Security

YamJam data is secured in the same method your .ssh directory is secured, *chmod 700*. It doesn't support encryption intrinsically. It can return an encrypted value your have put there but that is up to your app. So follow the install instructions and *chmod 700* your ~/.yamjam directory.

### 3.4.4 What is so special about YamJam, I could roll my own

Not a thing is technically special about YamJam. Yes, it's got a decent mergeDict routine. The problem with rolling your own is the same type of problem as rolling your own templating system. Why not just *pip install yamjam* instead of maintaining the code yourself. That is the thing about YamJam, it's easy to install. It takes the mundane task and rolls it up so you can write more exciting code. Just for the mere fact you can simply install it and then start factoring out sensitive and installation dependent data that has leaked into your repos.

### 3.4.5 What's up with the name, YamJam?

One day I was sitting there, dealing with an issue of config differences between dev and production and I said to myself, wtf? Why can't I just take my config settings and jam it in a yaml file? Hence, YamJam.

## 3.5 Contributing

You can contribute to the project in a number of ways. Code is always good, bugs are interesting but telling a friend or a peer about YamJam is awsome.

### 3.5.1 Code

1. Fork the repository on Bitbucket .
2. Make a virtualenv, clone the repos, install the deps from *pip install -r requirements-dev.txt*
3. Write any new tests needed and ensure existing tests continue to pass without modification.
1. Setup CI testing on drone.io for your Fork. See current script .
4. Ensure that your name is added to the end of the *Authors* file using the format Name <email@domain.com> (url), where the (url) portion is optional.
5. Submit a Pull Request to the project on Bitbucket.

### 3.5.2 Bug Reports

If you encounter a bug or some surprising behavior, please file an issue on our tracker

### 3.5.3 Get the Word Out

- Tell a friend
- email a list
- blog about about it
- give a talk to your local users group

Let's face it, a config system is not that exciting (unless you get it wrong - Windows Registry). But just like templating, everyone needs it. You roll your own, you try ton's of if/then and environment detection but it always ends up lacking. YamJam makes it sane and installable from pypi.

Now instead of spending attention cycles on your own boring implementation, you can spend your time on interesting code and let us take of the mundane.

## 3.6 Release Procedures

Steps necessary for an uneventful release of YamJam.

### 3.6.1 Full Release

1. Merge in outstanding Pull requests
2. Ensure test suite pass / drone.io

3. run *release.sh –dry-run* and ensure all looks well
4. run *release.sh* and make sure to push any outstanding commits to master
5. create a new empty environment and *pip install /localpath/yamjam-x.y.z.gz*
6. copy runtests.py to the environment created in previous step and run it
7. upload package to pypi, *twine upload sdist/**
8. verify new dist is available on pypi
9. Be Happy

### 3.6.2 Build Dist Only

```
python setup.py sdist
```

### 3.6.3 Install and test dist locally

1. Build Dist Only
2. Create clean environment
3. install local dist file in clean environment

```
pip install /localpath/to/dist/yamjam-x.y.z.gz
```

4. copy runtests.py to clean environment and run it
5. remove testing environment

## 3.7 Authors

Contributors of code, tests and documentation to the project who have agreed to have their work enjoined into the project and project license (BSD).

- Jeff Hinrichs <dundeemt@gmail.com>, http://inre.dundeemt.com/

CHAPTER 4

# Indices and tables

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