Everett Documentation

Release 1.0.0 20190107

Will Kahn-Greene

Jan 07, 2019

Contents

1 Goals **3**

2 Why not other libs? **5**

3 Quick start **7**
- 3.1 Fast start example . . . 7
- 3.2 More complex example . . . 7
- 3.3 Wrapping configuration in a configuration class . . . 9
- 3.4 Everett components . . . 10

4 Install **13**
- 4.1 Install from PyPI . . . 13
- 4.2 Install for hacking . . . 13

5 Contents **15**
- 5.1 History . . . 15
- 5.2 Configuration . . . 18
- 5.3 Components . . . 35
- 5.4 Using the Sphinx extension . . . 36
- 5.5 Recipes . . . 38
- 5.6 Library . . . 43
- 5.7 Hacking . . . 50

6 Indices and tables **51**

Python Module Index **53**

Everett is a Python configuration library for your app.

Code https://github.com/willkg/everett

Issues https://github.com/willkg/everett/issues

License MPL v2

Documentation https://everett.readthedocs.io/

CHAPTER 1

Goals

Goals of Everett:

1. flexible configuration from multiple configured environments
2. easy testing with configuration
3. easy documentation of configuration for users

From that, Everett has the following features:

- is composeable and flexible
- makes it easier to provide helpful error messages for users trying to configure your software
- supports auto-documentation of configuration with a Sphinx `autocomponent` directive
- has an API for testing configuration variations in your tests
- can pull configuration from a variety of specified sources (environment, ini files, dict, write-your-own)
- supports parsing values (bool, int, lists of things, classes, write-your-own)
- supports key namespaces
- supports component architectures
- works with whatever you're writing–command line tools, web sites, system daemons, etc

Everett is inspired by python-decouple and configman.

CHAPTER 2

Why not other libs?

Most other libraries I looked at had one or more of the following issues:

- were tied to a specific web app framework
- didn't allow you to specify configuration sources
- provided poor error messages when users configure things wrong
- had a global configuration object
- made it really hard to override specific configuration when writing tests
- had no facilities for auto-generating configuration documentation

CHAPTER 3

Quick start

3.1 Fast start example

You have an app and want it to look for configuration first in an `.env` file in the current working directory, then then in the process environment. You can do this:

```
from everett.manager import ConfigManager

config = ConfigManager.basic_config()
```

Then you can use it like this:

```
debug_mode = config('debug', parser=bool)
```

When you outgrow that or need different variations of it, you can change that to creating a `ConfigManager` from scratch.

3.2 More complex example

We have an app and want to pull configuration from an INI file stored in a place specified by `MYAPP_INI` in the environment, `~/.myapp.ini`, or `/etc/myapp.ini` in that order.

We want to pull infrastructure values from the environment.

Values from the environment should override values from the INI file.

First, we need to install the additional requirements for INI file environments:

```
pip install everett[ini]
```

Then we set up our `ConfigManager`:

```
import os
import sys

from everett.ext.inifile import ConfigIniEnv
from everett.manager import ConfigManager, ConfigOSEnv


def get_config():
    return ConfigManager(
        # Specify one or more configuration environments in
        # the order they should be checked
        environments=[
            # Look in OS process environment first
            ConfigOSEnv(),

            # Look in INI files in order specified
            ConfigIniEnv([
                os.environ.get('MYAPP_INI'),
                '~/.myapp.ini',
                '/etc/myapp.ini'
            ]),
        ],

        # Provide users a link to documentation for when they hit
        # configuration errors
        doc='Check https://example.com/configuration for docs.'
    )
```

Then we use it:

```
def is_debug(config):
    return config('debug', parser=bool,
        doc='Switch debug mode on and off.')


config = get_config()

if is_debug(config):
    print('DEBUG MODE ON!')
```

Let's write some tests that verify behavior based on the `debug` configuration value:

```
from myapp import get_config, is_debug

from everett.manager import config_override


@config_override(DEBUG='true')
def test_debug_true():
    assert is_debug(get_config()) is True


@config_override(DEBUG='false')
def test_debug_false():
    assert is_debug(get_config()) is False
```

If the user sets `DEBUG` wrong, they get a helpful error message with the documentation for the configuration option and the `ConfigManager`:

```
$ DEBUG=foo python myprogram.py
<traceback>
namespace=None key=debug requires a value parseable by bool
Switch debug mode on and off.
Check https://example.com/configuration for docs.
```

3.3 Wrapping configuration in a configuration class

Everett supports wrapping your configuration in an instance. Let's rewrite the above example using a configuration class.

First, create a configuration class:

```
import os
import sys

from everett.component import RequiredConfigMixin, ConfigOptions
from everett.ext.inifile import ConfigIniEnv
from everett.manager import ConfigManager, ConfigOSEnv


class AppConfig(RequiredConfigMixin):
    required_config = ConfigOptions()
    required_config.add_option(
        'debug',
        parser=bool,
        default='false',
        doc='Switch debug mode on and off.')
    )
```

Then we set up our `ConfigManager`:

```
def get_config():
    manager = ConfigManager(
        # Specify one or more configuration environments in
        # the order they should be checked
        environments=[
            # Look in OS process environment first
            ConfigOSEnv(),

            # Look in INI files in order specified
            ConfigIniEnv([
                os.environ.get('MYAPP_INI'),
                '~/.myapp.ini',
                '/etc/myapp.ini'
            ]),
        ],

        # Provide users a link to documentation for when they hit
        # configuration errors
        doc='Check https://example.com/configuration for docs.'
    )

    # Bind the manager to the configuration class
    return manager.with_options(AppConfig())
```

Then use it:

```
config = get_config()

if config('debug'):
    print('DEBUG MODE ON!')
```

Further, you can auto-generate configuration documentation by including the `everett.sphinxext` Sphinx extension and using the `autocomponent` directive:

```
.. autocomponent:: path.to.AppConfig
```

That kind of looks the same, but it has a few niceties:

1. your application configuration is centralized in one place instead of spread out across your code base
2. you can use the `autocomponent` Sphinx directive to auto-generate configuration documentation for your users

3.4 Everett components

Everett supports components. Say your app needs to connect to RabbitMQ. With Everett, you can wrap the configuration up with the component:

```
from everett.component import RequiredConfigMixin, ConfigOptions


class RabbitMQComponent(RequiredConfigMixin):
    required_config = ConfigOptions()
    required_config.add_option(
        'host',
        doc='RabbitMQ host to connect to'
    )
    required_config.add_option(
        'port',
        default='5672',
        doc='Port to use',
        parser=int
    )
    required_config.add_option(
        'queue_name',
        doc='Queue to insert things into'
    )

    def __init__(self, config):
        # Bind the configuration to just the configuration this
        # component requires such that this component is
        # self-contained.
        self.config = config.with_options(self)

        self.host = self.config('host')
        self.port = self.config('port')
        self.queue_name = self.config('queue_name')
```

Then instantiate a `RabbitMQComponent`, but with configuration in the `rmq` namespace:

```
queue = RabbitMQComponent(config.with_namespace('rmq'))
```

In your environment, you would provide `RMQ_HOST`, etc for this component.

You can auto-generate configuration documentation for this component in your Sphinx docs by including the `everett.sphinxext` Sphinx extension and using the `autocomponent` directive:

```
.. autocomponent:: path.to.RabbitMQComponent
```

Say your app actually needs to connect to two separate queues–one for regular processing and one for priority processing:

```
regular_queue = RabbitMQComponent(
    config.with_namespace('regular').with_namespace('rmq')
)
priority_queue = RabbitMQComponent(
    config.with_namespace('priority').with_namespace('rmq')
)
```

In your environment, you provide the regular queue configuration with `RMQ_REGULAR_HOST`, etc and the priority queue configuration with `RMQ_PRIORITY_HOST`, etc.

Same component code. Two different instances pulling configuration from two different namespaces.

Components support subclassing, mixins and all that, too.

CHAPTER 4

Install

4.1 Install from PyPI

Run:

```
$ pip install everett
```

If you want to use the `ConfigIniEnv`, you need to install its requirements as well:

```
$ pip install everett[ini]
```

4.2 Install for hacking

Run:

```
# Clone the repository
$ git clone https://github.com/willkg/everett

# Create a virtualenvironment
$ mkvirtualenv --python /usr/bin/python3 everett
...

# Install Everett and dev requirements
$ pip install -r requirements-dev.txt
```

CHAPTER 5

Contents

5.1 History

5.1.1 1.0.0 (January 7th, 2019)

Backwards incompatible changes:

- Dropped support for Python 2.7. Everett no longer supports Python 2. (#73)
- Dropped support for Python 3.3 and added support for Python 3.7. Thank you, pjz! (#68)
- Moved `ConfigIniEnv` to a different module. Now you need to import it like this:

  ```
  from everett.ext.inifile import ConfigIniEnv
  ```

  (#79)

Features:

- Everett now logs configuration discovery in the `everett` logger at the `logging.DEBUG` level. This is helpful for trouble-shooting some kinds of issues. (#74)

Fixes:

- Everett no longer requires `configobj`–it's now optional. If you use `ConfigIniEnv`, you can install it with:

  ```
  $ pip install everett[ini]
  ```

  (#79)

- Fixed list parsing and file discovery in ConfigIniEnv so they match the docs and are more consistent with other envs. Thank you, apollo13! (#71)
- Added a `.basic_config()` for fast opinionated setup that uses the process environment and a `.env` file in the current working directory.
- Switching to semver.

5.1.2 0.9 (April 7th, 2017)

Changed:

- Rewrite Sphinx extension. The extension is now in the `everett.sphinxext` module and the directive is now `.. autocomponent::`. It generates better documentation and it now indexes Everett components and options.

  This is backwards-incompatible. You will need to update your Sphinx configuration and documentation.

- Changed the `HISTORY.rst` structure.
- Changed the repr for `everett.NO_VALUE` to `"NO_VALUE"`.
- `InvalidValueError` and `ConfigurationMissingError` now have `namespace`, `key`, and `parser` attributes allowing you to build your own messages.

Fixed:

- Fix an example in the docs where the final key was backwards. Thank you, pjz!

Documentation fixes and updates.

5.1.3 0.8 (January 24th, 2017)

Added:

- Add `:namespace:` and `:case:` arguments to autoconfig directive. These make it easier to cater your documentation to your project's needs.
- Add support for Python 3.6.

Minor documentation fixes and updates.

5.1.4 0.7 (January 5th, 2017)

Added:

- Feature: You can now include documentation hints and urls for `ConfigManager` objects and config options. This will make it easier for your users to debug configuration errors they're having with your software.

Fixed:

- Fix `ListOf` so it returns empty lists rather than a list with a single empty string.

Documentation fixes and updates.

5.1.5 0.6 (November 28th, 2016)

Added:

- Add `RequiredConfigMixin.get_runtime_config()` which returns the runtime configuration for a component or tree of components. This lets you print runtime configuration at startup, generate INI files, etc.
- Add `ConfigObjEnv` which lets you use an object for configuration. This works with argparse's Namespace amongst other things.

Changed:

- Change `:show-docstring:` to take an optional value which is the attribute to pull docstring content from. This means you don't have to mix programming documentation with user documentation–they can be in different attributes.
- Improve configuration-related exceptions. With Python 3, configuration errors all derive from `ConfigurationError` and have helpful error messages that should make it clear what's wrong with the configuration value. With Python 2, you can get other kinds of Exceptions thrown depending on the parser used, but configuration error messages should still be helpful.

Documentation fixes and updates.

5.1.6 0.5 (November 8th, 2016)

Added:

- Add `:show-docstring:` flag to `autoconfig` directive.
- Add `:hide-classname:` flag to `autoconfig` directive.

Changed:

- Rewrite `ConfigIniEnv` to use configobj which allows for nested sections in INI files. This also allows you to specify multiple INI files and have later ones override earlier ones.

Fixed:

- Fix `autoconfig` Sphinx directive and add tests–it was all kinds of broken.

Documentation fixes and updates.

5.1.7 0.4 (October 27th, 2016)

Added:

- Add `raw_value` argument to config calls. This makes it easier to write code that prints configuration.

Fixed:

- Fix `listify(None)` to return `[]`.

Documentation fixes and updates.

5.1.8 0.3.1 (October 12th, 2016)

Fixed:

- Fix `alternate_keys` with components. Previously it worked for everything but components. Now it works with components, too.

Documentation fixes and updates.

5.1.9 0.3 (October 6th, 2016)

Added:

- Add `ConfigManager.from_dict()` shorthand for building configuration instances.
- Add `.get_namespace()` to `ConfigManager` and friends for getting the complete namespace for a given config instance as a list of strings.

- Add `alternate_keys` to config call. This lets you specify a list of keys in order to try if the primary key doesn't find a value. This is helpful for deprecating keys that you used to use in a backwards-compatible way.
- Add `root:` prefix to keys allowing you to look outside of the current namespace and at the configuration root for configuration values.

Changed:

- Make `ConfigDictEnv` case-insensitive to keys and namespaces.

Documentation fixes and updates.

5.1.10 0.2 (August 16th, 2016)

Added:

- Add `ConfigEnvFileEnv` for supporting `.env` files. Thank you, Paul!
- Add "on" and "off" as valid boolean values. This makes it easier to use config for feature flippers. Thank you, Paul!

Changed:

- Change `ConfigIniEnv` to take a single path or list of paths. Thank you, Paul!
- Make `NO_VALUE` falsy.

Fixed:

- Fix `__call__` returning None–it should return `NO_VALUE`.

Lots of docs updates: finished the section about making your own parsers, added a section on using dj-database-url, added a section on django-cache-url and expanded on existing examples.

5.1.11 0.1 (August 1st, 2016)

Initial writing.

5.2 Configuration

Contents

- *Configuration*
  - *Setting up configuration in your app*
    - *Create a ConfigManager and specify sources*
    - *Specify pointer to configuration errors docs*
  - *Where to put ConfigManager*
  - *Configuration sources*
    - *Dict (ConfigDictEnv)*
    - *Process environment (ConfigOSEnv)*
    - *ENV files (ConfigEnvFileEnv)*

        - *Python objects (ConfigObjEnv)*
        - *INI files (ConfigIniEnv)*
        - *YAML files (ConfigYamlEnv)*
        - *Implementing your own configuration environments*
    - *Extracting values*
    - *Handling exceptions when extracting values*
    - *Namespaces*
    - *Parsers*
        - *Python types like str, int, float, pathlib.Path*
        - *bools*
        - *classes*
        - *ListOf(parser)*
        - *dj_database_url*
        - *django-cache-url*
        - *Implementing your own parsers*
    - *Trouble-shooting and logging what happened*

5.2.1 Setting up configuration in your app

Create a ConfigManager and specify sources

Configuration is handled by a `ConfigManager`. When you instantiate the `ConfigManager`, you pass it a list of sources that it should look at when resolving configuration requests. The list of sources are consulted in the order you specify.

For example:

```
import os
from everett.ext.inifile import ConfigIniEnv
from everett.manager import (
    ConfigDictEnv,
    ConfigManager,
    ConfigOSEnv
)

config = ConfigManager([
    # Pull from the OS environment first
    ConfigOSEnv(),

    # Fall back to the file specified by the FOO_INI OS environment
    # variable if such file exists
    ConfigIniEnv(os.environ.get('FOO_INI')),

    # Fall back to this dict of defaults
    ConfigDictEnv({
        'FOO_VAR': 'bar'
```

(continues on next page)

(continued from previous page)

```
    })
])

assert config('foo_var') == 'bar'
```

Specify pointer to configuration errors docs

In addition to a list of sources, you can provide a `doc`. You can use this to guide users trying to use your software and hitting configuration errors to documentation for your configuration.

Here's a trivial program:

```
from everett.manager import ConfigManager, ConfigOSEnv


def main():
    config = ConfigManager(
        environments=[ConfigOSEnv()],
        doc='For configuration help, see https://example.com/configuration'
    )

    debug_mode = config(
        'debug', default='false', parser=bool,
        doc='True to put the app in debug mode. Don\'t use this in production!'
    )

    if debug_mode:
        print('Debug mode is on!')
    else:
        print('Debug mode off.')


if __name__ == '__main__':
    main()
```

Let's configure the program wrong and run it in Python 3 and see what it tells us:

```
$ python trivial.py
Debug mode off.

$ DEBUG=true python trivial.py
Debug mode is on!

$ DEBUG=omg python trivial.py
Traceback (most recent call last):
  File "/home/willkg/mozilla/everett/everett/manager.py", line 908, in __call__
    return parser(val)
  File "/home/willkg/mozilla/everett/everett/manager.py", line 109, in parse_bool
    raise ValueError('"%s" is not a valid bool value' % val)
ValueError: "omg" is not a valid bool value

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "configuration_doc.py", line 22, in <module>
```

(continues on next page)

(continued from previous page)

```
    main()
  File "configuration_doc.py", line 12, in main
    doc='True to put the app in debug mode. Don\'t use this in production!'
  File "/home/willkg/mozilla/everett/everett/manager.py", line 936, in __call__
    raise InvalidValueError(msg)
everett.InvalidValueError: ValueError: "omg" is not a valid bool value
namespace=None key=debug requires a value parseable by everett.manager.parse_bool
True to put the app in debug mode. Don't use this in production!
For configuration help, see https://example.com/configuration
```

Here, we see the documentation for the configuration item, the documentation from the `ConfigManager` and the specific Python exception information.

5.2.2 Where to put ConfigManager

You can create the `ConfigManager` when instantiating the app–that works fine. You could also create it as a global singleton.

`ConfigManager` should be thread-safe and re-entrant with the provided sources. If you implement your own configuration environments, then thread-safety and re-entrantcy depend on whether your configuration environments are safe in these ways.

5.2.3 Configuration sources

Dict (ConfigDictEnv)

class everett.manager.**ConfigDictEnv**(*cfg*)

Source for pulling configuration out of a dict

This is handy for testing. You might also use it if you wanted to move all your defaults values into one centralized place.

Keys are prefixed by namespaces and the whole thing is uppercased.

For example, namespace "bar" for key "foo" becomes `BAR_FOO` in the dict.

For example:

```
from everett.manager import ConfigDictEnv, ConfigManager

config = ConfigManager([
    ConfigDictEnv({
        'FOO_BAR': 'someval',
        'BAT': '1',
    })
])
```

Keys are not case sensitive. This also works:

```
from everett.manager import ConfigDictEnv, ConfigManager

config = ConfigManager([
    ConfigDictEnv({
        'foo_bar': 'someval',
        'bat': '1',
```

(continues on next page)

(continued from previous page)

```
    })
])

print config('foo_bar')
print config('FOO_BAR')
print config.with_namespace('foo')('bar')
```

Also, `ConfigManager` has a convenience classmethod for creating a `ConfigManager` with just a dict environment:

```
from everett.manager import ConfigManager

config = ConfigManager.from_dict({
    'FOO_BAR': 'bat'
})
```

Changed in version 0.3: Keys are no longer case-sensitive.

Process environment (ConfigOSEnv)

class everett.manager.**ConfigOSEnv**

Source for pulling configuration out of the environment

This source lets you specify configuration in the environment. This is useful for infrastructure related configuration like usernames and ports and secret configuration like passwords.

Keys are prefixed by namespaces and the whole thing is uppercased.

For example, key "foo" will be `FOO` in the environment.

For example, namespace "bar" for key "foo" becomes `BAR_FOO` in the environment.

Key and namespace can consist of alphanumeric characters and _.

Note: Unlike other config environments, this one is case sensitive in that keys defined in the environment **must** be all uppercase.

For example, these are good:

```
FOO=bar
FOO_BAR=bar
FOO_BAR1=bar
```

This is bad:

```
foo=bar
```

To use, instantiate and toss in the source list:

```
from everett.manager import ConfigOSEnv, ConfigManager

config = ConfigManager([
    ConfigOSEnv()
])
```

ENV files (ConfigEnvFileEnv)

class everett.manager.**ConfigEnvFileEnv**(*possible_paths*)

Source for pulling configuration out of .env files

This source lets you specify configuration in an .env file. This is useful for local development when in production you use values in environment variables.

Keys are prefixed by namespaces and the whole thing is uppercased.

For example, key "foo" will be `FOO` in the file.

For example, namespace "bar" for key "foo" becomes `BAR_FOO` in the file.

Key and namespace can consist of alphanumeric characters and _.

To use, instantiate and toss in the source list:

```
from everett.manager import ConfigEnvFileEnv, ConfigManager

config = ConfigManager([
    ConfigEnvFileEnv('.env')
])
```

For multiple paths:

```
from everett.manager import ConfigEnvFileEnv, ConfigManager

config = ConfigManager([
    ConfigEnvFileEnv([
        '.env',
        'config/prod.env'
    ])
])
```

Here's an example .env file:

```
DEBUG=true

# secrets
SECRET_KEY=ou812

# database setup
DB_HOST=localhost
DB_PORT=5432
```

Python objects (ConfigObjEnv)

class everett.manager.**ConfigObjEnv**(*obj*, *force_lower=True*)

Source for pulling configuration values out of a Python object

This is handy for a few weird situations. For example, you can use this to "bridge" Everett configuration with command line arguments. The argparse Namespace works fine here.

Namespace (the Everett one–not the argparse one) is prefixed. So key "foo" in namespace "bar" is "foo_bar".

For example:

```
import argparse

from everett.manager import ConfigObjEnv, ConfigManager

parser = argparse.ArgumentParser()
parser.add_argument(
    '--debug', help='to debug or not to debug'
)
parsed_vals = parser.parse_known_args()[0]

config = ConfigManager([
    ConfigObjEnv(parsed_vals)
])

print config('debug', parser=bool)
```

Keys are not case-sensitive–everything is converted to lowercase before pulling it from the object.

Note: ConfigObjEnv has nothing to do with the library configobj.

New in version 0.6.

INI files (ConfigIniEnv)

class everett.ext.inifile.**ConfigIniEnv**(*possible_paths*)

Source for pulling configuration from INI files

This requires optional dependencies. You can install them with:

```
$ pip install everett[ini]
```

Takes a path or list of possible paths to look for a INI file. It uses the first INI file it can find.

If it finds no INI files in the possible paths, then this configuration source will be a no-op.

This will expand ~ as well as work relative to the current working directory.

This example looks just for the INI file specified in the environment:

```
from everett.manager import ConfigManager
from everett.ext.inifile import ConfigIniEnv

config = ConfigManager([
    ConfigIniEnv(os.environ.get('FOO_INI'))
])
```

If there's no `FOO_INI` in the environment, then the path will be ignored.

Here's an example that looks for the INI file specified in the environment variable `FOO_INI` and failing that will look for `.antenna.ini` in the user's home directory:

```
from everett.manager import ConfigManager
from everett.ext.inifile import ConfigIniEnv

config = ConfigManager([
    ConfigIniEnv([
```

(continues on next page)

(continued from previous page)

```
        os.environ.get('FOO_INI'),
        '~/.antenna.ini'
    ])
])
```

This example looks for a `config/local.ini` file which overrides values in a `config/base.ini` file both are relative to the current working directory:

```
from everett.manager import ConfigManager
from everett.ext.inifile import ConfigIniEnv

config = ConfigManager([
    ConfigIniEnv('config/local.ini'),
    ConfigIniEnv('config/base.ini')
])
```

Note how you can have multiple `ConfigIniEnv` files and this is how you can set Everett up to have values in one INI file override values in another INI file.

INI files must have a "main" section. This is where keys that aren't in a namespace are placed.

Minimal INI file:

```
[main]
```

In the INI file, namespace is a section. So key "user" in namespace "foo" is:

```
[foo]
user=someval
```

Everett uses configobj, so it supports nested sections like this:

```
[main]
foo=bar

[namespace]
foo2=bar2

  [[namespace2]]
  foo3=bar3
```

Which gives you these:

- `FOO`
- `NAMESPACE_FOO2`
- `NAMESPACE_NAMESPACE2_FOO3`

See more details here: http://configobj.readthedocs.io/en/latest/configobj.html#the-config-file-format

YAML files (ConfigYamlEnv)

class everett.ext.yamlfile.**ConfigYamlEnv**(*possible_paths*)

Source for pulling configuration from YAML files

This requires optional dependencies. You can install them with:

```
$ pip install everett[yaml]
```

Takes a path or list of possible paths to look for a YAML file. It uses the first YAML file it can find.

If it finds no YAML files in the possible paths, then this configuration source will be a no-op.

This will expand ~ as well as work relative to the current working directory.

This example looks just for the YAML file specified in the environment:

```
from everett.manager import ConfigManager
from everett.ext.yamlfile import ConfigYamlEnv

config = ConfigManager([
    ConfigYamlEnv(os.environ.get('FOO_YAML'))
])
```

If there's no `FOO_YAML` in the environment, then the path will be ignored.

Here's an example that looks for the YAML file specified in the environment variable `FOO_YAML` and failing that will look for `.antenna.yaml` in the user's home directory:

```
from everett.manager import ConfigManager
from everett.ext.yamlfile import ConfigYamlEnv

config = ConfigManager([
    ConfigYamlEnv([
        os.environ.get('FOO_YAML'),
        '~/.antenna.yaml'
    ])
])
```

This example looks for a `config/local.yaml` file which overrides values in a `config/base.yaml` file both are relative to the current working directory:

```
from everett.manager import ConfigManager
from everett.ext.yamlfile import ConfigYamlEnv

config = ConfigManager([
    ConfigYamlEnv('config/local.yaml'),
    ConfigYamlEnv('config/base.yaml')
])
```

Note how you can have multiple `ConfigYamlEnv` files. This is how you can set Everett up to have values in one YAML file override values in another YAML file.

Everett looks for keys and values in YAML files. YAML files can be split into multiple documents, but Everett only looks at the first one.

Keys are case-insensitive. You can do namespaces either in the key itself using _ as a separator or as nested mappings.

All values should be double-quoted.

Here's an example:

```
foo: "bar"
FOO2: "bar"
namespace_foo: "bar"
namespace:
```

(continues on next page)

(continued from previous page)

```
    namespace2:
        foo: "bar"
```

Giving you these namespaced keys:

- FOO
- FOO2
- NAMESPACE_FOO
- NAMESPACE_NAMEPSACE2_FOO

Implementing your own configuration environments

You can implement your own configuration environments. For example, maybe you want to pull configuration from a database or Redis or a post-it note on the refrigerator.

They just need to implement the `.get()` method. A no-op implementation is this:

```
from everett import NO_VALUE
from everett.manager import listify


class NoOpEnv(object):
    def get(self, key, namespace=None):
        # The namespace is either None, a string or a list of
        # strings. This converts it into a list.
        namespace = listify(namespace)

        # FIXME: Your code to extract the key in namespace here.

        # At this point, the key doesn't exist in the namespace
        # for this environment, so return a ``NO_VALUE``.
        return NO_VALUE
```

Generally, environments should return a value if the key exists in that environment and should return `NO_VALUE` if and only if the key does not exist in that environment.

For exceptions, it depends on what you want to have happen. It's ok to let exceptions go unhandled–Everett will wrap them in a `ConfigurationError`. If your environment promises never to throw an exception, then you should handle them all and return `NO_VALUE` since with that promise all exceptions would indicate the key is not in the environment.

5.2.4 Extracting values

Once you have a configuration manager set up with sources, you can pull configuration values from it.

Configuration must have a key. Other than that, everything is optionally specified.

`ConfigManager.__call__`(*key*, *namespace=None*, *default=NO_VALUE*, *alternate_keys=NO_VALUE*, *doc=''*, *parser=<class 'str'>*, *raise_error=True*, *raw_value=False*)

Returns a parsed value from the environment

Parameters

- **key** – the key to look up

- **namespace** – the namespace for the key–different environments use this differently
- **default** – the default value (if any); this must be a string that is parseable by the specified parser; if no default is provided, this will raise an error or return `everett.NO_VALUE` depending on the value of `raise_error`
- **alternate_keys** – the list of alternate keys to look up; supports a `root:` key prefix which will cause this to look at the configuration root rather than the current namespace

  New in version 0.3.
- **doc** – documentation for this config option

  New in version 0.6.
- **parser** – the parser for converting this value to a Python object
- **raise_error** – True if you want a lack of value to raise a `everett.ConfigurationError`
- **raw_value** – True if you want the raw unparsed value, False otherwise

Raises

- ***everett.ConfigurationMissingError*** – if the required bit of configuration is missing from all the environments
- ***everett.InvalidKeyError*** – if the configuration key doesn't exist for that component
- **everet.InvalidValueError** – (Python 3-only) if the configuration value is invalid in some way (not an integer, not a bool, etc)
- **Exception subclass** – (Python 2-only) parser code can raise anything and since this is Python 2, we can't do much about it without stomping on the traceback so we change the message and raise the same exception

Examples:

```
config = ConfigManager([])

# Use the special bool parser
DEBUG = config('DEBUG', default='false', parser=bool)
DEBUG = config('DEBUG', default='True', parser=bool)
DEBUG = config('DEBUG', default='true', parser=bool)
DEBUG = config('DEBUG', default='yes', parser=bool)
DEBUG = config('DEBUG', default='y', parser=bool)

# Use the list of parser
from everett.manager import ListOf
ALLOWED_HOSTS = config('ALLOWED_HOSTS', default='localhost',
                       parser=ListOf(str))

# Use alternate_keys for backwards compatibility with an
# older version of this software
PASSWORD = config('PASSWORD', alternate_keys=['SECRET'])
```

The default value should **always** be a string that is parseable by the parser. This simplifies thinking about values since **all** values are strings that are parsed by the parser rather than default values do one thing and non-default values doa nother. Further, it simplifies documentation for the user since the default value is an example value.

The parser can be any callable that takes a string value and returns a parsed value.

Some more examples:

config('password') The key is "password".

The value is parsed as a string.

There is no default value provided so if "password" isn't provided in any of the configuration sources, then this will raise a `everett.ConfigurationError`.

This is what you want to do to require that a configuration value exist.

config('name', raise_error=False) The key is "name".

The value is parsed as a string.

There is no default value provided and raise_error is set to False, so if this configuration variable isn't set anywhere, the result of this will be `everett.NO_VALUE`.

Note: `everett.NO_VALUE` is a falsy value so you can use it in comparative contexts:

```
debug = config('DEBUG', parser=bool, raise_error=False)
if not debug:
    pass
```

---

config('debug', default='false', parser=bool) The key is "debug".

The value is parsed using the special Everett bool parser.

There is a default provided, so if this configuration variable isn't set in the specified sources, the default will be false.

config('username', namespace='db') The key is "username".

The namespace is "db".

There's no default, so if there's no "username" in namespace "db" configuration variable set in the sources, this will raise a `everett.ConfigurationError`.

config('password', namespace='postgres', alternate_keys=['db_password', 'root:postgres_pass
The key is "password".

The namespace is "postgres".

If there is no key "password" in namespace "postgres", then it looks for "db_password" in namespace "postgres". This makes it possible to deprecate old key names, but still support them.

If there is no key "password" or "db_password" in namespace "postgres", then it looks at "postgres_password" in the root namespace. This allows you to have multiple components that share configuration like credentials and hostnames.

config('port', parser=int, doc='The port you want this to listen on.') You can provide a `doc` argument which will give users users who are trying to configure your software a more helpful error message when they hit a configuration error.

Example of error message with `doc`:

```
everett.InvalidValueError: ValueError: invalid literal for int() with base 10:
'bar'; namespace=None key=foo requires a value parseable by int
The port you want this to listen on.
```

That last line comes directly from the `doc` argument you provide.

class everett.**ConfigurationError**
Configuration error base class

class everett.**InvalidValueError**(**args*, ***kwargs*)
Indicates that the value is not valid

class everett.**ConfigurationMissingError**(**args*, ***kwargs*)
Indicates that required configuration is missing

class everett.**InvalidKeyError**
Indicates the key is not valid for this component

5.2.5 Handling exceptions when extracting values

Getting configuration should always return a subclass of *everett.ConfigurationError*. This makes it easier to programmatically figure out what happened.

For example:

```
#!/usr/bin/env python3

import logging

from everett import InvalidValueError
from everett.manager import ConfigManager

logging.basicConfig()

config = ConfigManager.from_dict({
    'debug_mode': 'monkey'
})

try:
    some_val = config('debug_mode', parser=bool)
except InvalidValueError:
    # The "debug_mode" configuration value is incorrect--alert
    # user in the logs.
    logging.exception('gah!')
```

That logs this:

```
ERROR:root:gah!
Traceback (most recent call last):
  File "/home/willkg/mozilla/everett/everett/manager.py", line 908, in __call__
    return parser(val)
  File "/home/willkg/mozilla/everett/everett/manager.py", line 109, in parse_bool
    raise ValueError('"%s" is not a valid bool value' % val)
ValueError: "monkey" is not a valid bool value

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "configuration_handling_exceptions.py", line 13, in <module>
    some_val = config('debug_mode', parser=bool)
  File "/home/willkg/mozilla/everett/everett/manager.py", line 936, in __call__
    raise InvalidValueError(msg)
everett.InvalidValueError: ValueError: "monkey" is not a valid bool value
namespace=None key=debug_mode requires a value parseable by everett.manager.parse_bool
```

5.2.6 Namespaces

Everett has namespaces for grouping related configuration values.

For example, say you had database code that required a username, password and port. You could do something like this:

```
def open_db_connection(config):
    username = config('username', namespace='db')
    password = config('password', namespace='db')
    port = config('port', namespace='db', default=5432, parser=int)


conn = open_db_connection(config)
```

These variables in the environment would be `DB_USERNAME`, `DB_PASSWORD` and `DB_PORT`.

This is helpful when you need to create two of the same thing, but using separate configuration. Extending this example, you could pass the namespace as an argument.

For example, say you wanted to use `open_db_connection` for a source db and for a dest db:

```
def open_db_connection(config, namespace):
    username = config('username', namespace=namespace)
    password = config('password', namespace=namespace)
    port = config('port', namespace=namespace, default=5432, parser=int)


source = open_db_connection(config, 'source_db')
dest = open_db_connection(config, 'dest_db')
```

Then you end up with `SOURCE_DB_USERNAME` and friends and `DEST_DB_USERNAME` and friends.

5.2.7 Parsers

Python types like str, int, float, pathlib.Path

Python types can convert strings to Python values. You can use these as parsers:

- `str`
- `int`
- `float`
- `pathlib.Path`

bools

Everett provides a special bool parser that handles more explicit values for "true" and "false":

- true: t, true, yes, y, on, 1 (and uppercase versions)
- false: f, false, no, n, off, 0 (and uppercase versions)

`everett.manager.`**`parse_bool`**(*val*)
Parses a bool

Handles a series of values, but you should probably standardize on "true" and "false".

```
>>> parse_bool('y')
True
>>> parse_bool('FALSE')
False
```

classes

Everett provides a `everett.manager.parse_class` that takes a string specifying a module and class and returns the class.

`everett.manager.`**`parse_class`**(*val*)

Parses a string, imports the module and returns the class

```
>>> parse_class('hashlib.md5')
<built-in function openssl_md5>
```

ListOf(parser)

Everett provides a special `everett.manager.ListOf` parser which parses a list of some other type. For example:

```
ListOf(str)   # comma-delimited list of strings
ListOf(int)   # comma-delimited list of ints
```

`everett.manager.`**`ListOf`**(*parser*, *delimiter=', '*)

Parses a comma-separated list of things

```
>>> ListOf(str)('')
[]
>>> ListOf(str)('a,b,c,d')
['a', 'b', 'c', 'd']
>>> ListOf(int)('1,2,3,4')
[1, 2, 3, 4]
```

Note: This doesn't handle quotes or backslashes or any complicated string parsing.

For example:

```
>>> ListOf(str)('"a,b",c,d')
['"a', 'b"', 'c', 'd']
```

dj_database_url

Everett works great with dj-database-url.

For example:

```
import dj_database_url
from everett.manager import ConfigManager, ConfigOSEnv

config = ConfigManager([ConfigOSEnv()])
DATABASE = {
    'default': config('DATABASE_URL', parser=dj_database_url.parse)
}
```

That'll pull the `DATABASE_URL` value from the environment (it throws an error if it's not there) and runs it through `dj_database_url` which parses it and returns what Django needs.

With a default:

```
import dj_database_url
from everett.manager import ConfigManager, ConfigOSEnv

config = ConfigManager([ConfigOSEnv()])
DATABASE = {
    'default': config('DATABASE_URL', default='sqlite:///my.db',
                      parser=dj_database_url.parse)
}
```

Note: To use dj-database-url, you'll need to install it separately. Everett doesn't require it to be installed.

django-cache-url

Everett works great with django-cache-url.

For example:

```
import django_cache_url
from everett.manager import ConfigManager, ConfigOSEnv

config = ConfigManager([ConfigOSEnv()])
CACHES = {
    'default': config('CACHE_URL', parser=django_cache_url.parse)
}
```

That'll pull the `CACHE_URL` value from the environment (it throws an error if it's not there) and runs it through `django_cache_url` which parses it and returns what Django needs.

With a default:

```
import django_cache_url
from everett.manager import ConfigManager, ConfigOSEnv

config = ConfigManager([ConfigOSEnv()])
CACHES = {
    'default': config('CACHE_URL', default='locmem://myapp',
                      parser=django_cache_url.parse)
}
```

Note: To use django-cache-url, you'll need to install it separately. Everett doesn't require it to be installed.

Implementing your own parsers

It's easy to implement your own parser. You just need to build a callable that takes a string and returns the Python value you want.

If the value is not parseable, then it should raise a `ValueError`.

For example, say we wanted to implement a parser that returned yes/no/no-answer:

```
from everett.manager import ConfigManager


def parse_ynm(val):
    """Returns True, False or None (empty string)"""
    val = val.strip().lower()
    if not val:
        return None

    return val[0] == 'y'


config = ConfigManager.from_dict({
    'NO_ANSWER': '',
    'YES': 'yes',
    'ALSO_YES': 'y',
    'NO': 'no'
})

assert config('no_answer', parser=parse_ynm) is None
assert config('yes', parser=parse_ynm) is True
assert config('also_yes', parser=parse_ynm) is True
assert config('no', parser=parse_ynm) is False
```

Say you wanted to make a parser class that's line delimited:

```
from everett.manager import ConfigManager, get_parser


class Pairs(object):
    def __init__(self, val_parser):
        self.val_parser = val_parser

    def __call__(self, val):
        val_parser = get_parser(self.val_parser)
        out = []
        for part in val.split(','):
            k, v = part.split(':')
            out.append((k, val_parser(v)))
        return out


config = ConfigManager.from_dict({
    'FOO': 'a:1,b:2,c:3'
})

assert (
    config('FOO', parser=Pairs(int)) ==
    [('a', 1), ('b', 2), ('c', 3)]
)
```

5.2.8 Trouble-shooting and logging what happened

If you have a non-trivial Everett configuration, it might be difficult to figure out exactly why a key lookup failed.

Everett logs to the `everett` logger at the `logging.DEBUG` level. You can enable this logging and get a clearer idea of what's going on.

See Python logging documentation for details on enabling logging.

5.3 Components

Contents

- *Components*
  - *Building components with Everett*
  - *Getting configuration information for components*

5.3.1 Building components with Everett

Everett allows you to build components that specify their own configuration as a class property.

This lets you do three things:

1. instantiate components in a specified configuration namespace
2. restrict the configuration the component uses to that specified in the component
3. inherit and override configuration from superclasses

To create a component, you want to use the `everett.component.RequiredConfigMixin`.

For example, let's create a RabbitMQComponent for accessing RabbitMQ:

```
from everett.component import RequiredConfigMixin, ConfigOptions

class RabbitMQComponent(RequiredConfigMixin):
    required_config = ConfigOptions()
    required_config.add_option(
        'host',
        doc='RabbitMQ host to connect to'
    )
    required_config.add_option(
        'port',
        default='5672',
        doc='Port to use',
        parser=int
    )
    required_config.add_option(
        'queue_name',
        doc='Queue to insert things into'
    )

    def __init__(self, config):
        # Bind the configuration to just the configuration this
        # component requires such that this component is
        # self-contained.
        self.config = config.with_options(self)
        ...
```

`ConfigOptions` groups a set of configuration options that this component requires. Options of subclasses override those of superclasses.

We also need an `__init__` method that takes the `config` as an argument so that you can bind the component's config options with the config using `.with_options()`.

Then in our app, we instantiate a RabbitMQComponent, but with configuration in the rmq namespace:

```
queue = RabbitMQComponent(config.with_namespace('rmq'))
```

In our environment, we would provide `RMQ_HOST`, etc for this component.

Say our app actually needs to connect to two separate queues–one for regular processing and one for priority processing:

```
regular_queue = RabbitMQComponent(
    config.with_namespace('regular').with_namespace('rmq')
)
priority_queue = RabbitMQComponent(
    config.with_namespace('priority').with_namespace('rmq')
)
```

In our environment, we provide the regular queue configuration with `RMQ_REGULAR_HOST`, etc and the priority queue configuration with `RMQ_PRIORITY_HOST`, etc.

Same component code–two different instances.

5.3.2 Getting configuration information for components

The *everett.component.RequiredConfigMixin* has two methods to pull configuration about the component.

everett.component.RequiredConfigMixin.get_required_config() will return the required configuration for that component. This follows class hierarchies, handles configuration overrides and other things like that. The final *everett.component.ConfigOptions* class that you get is the configuration for that component.

See *everett.component.RequiredConfigMixin* for more details.

5.4 Using the Sphinx extension

No one likes to spend hours updating configuration documentation. Often, it's accidentally forgotten or overlooked when maintaining a project.

No one likes to spend hours trying to get something to work only to discover the configuration documentation is out of date, missing important information, or just wrong.

Blech.

Everett comes with a Sphinx extension to make it easier to document Everett components and configuration. This allows you to write configuration code and have it automatically documented in your Sphinx-generated docs without having to manually update it.

Further, Everett components will show up in the index making it easier for users to find what they're looking for.

Contains the autocomponent Sphinx extension for auto-documenting components with configuration.

The `autocomponent` declaration will pull out the class docstring as well as configuration requirements, throw it all in a blender and spit it out.

To configure Sphinx, add `'everett.sphinxext'` to the `extensions` in `conf.py`:

```
extensions = [
    ...
    'everett.sphinxext'
]
```

Note: You need to make sure that Everett is installed in the environment that Sphinx is being run in.

Use it like this in an `.rst` file to document a component:

```
.. autocomponent:: collector.ext.s3.S3CrashStorage
```

You can refer to that component in other parts of your docs and get a link by using the `:everett:component:` role:

```
Check out the :everett:component:`collector.ext.s3.S3CrashStorage`
configuration.
```

If your component class names are unique, then you can probably get away with:

```
Check out the :everett:component:`S3CrashStorage` configuration.
```

Changed in version 0.9: In Everett 0.8 and prior, the extension was in the `everett.sphinx_autoconfig` module and the directive was `.. autoconfig::`.

Showing docstring and content

If you want the docstring for the class, you can specify `:show-docstring:`:

```
.. autocomponent:: collector.external.boto.crashstorage.BotoS3CrashStorage
   :show-docstring:
```

If you want to show help, but from a different attribute than the docstring, you can specify any class attribute:

```
.. autocomponent:: collector.external.boto.crashstorage.BotoS3CrashStorage
   :show-docstring: __everett_help__
```

You can provide content as well:

```
.. autocomponent:: collector.external.boto.crashstorage.BotoS3CrashStorage

   This is some content!
```

New in version 0.5.

Hiding the class name

You can hide the class name if you want:

```
.. autocomponent:: collector.external.boto.crashstorage.BotoS3CrashStorage
   :hide-classname:
```

This is handy for application-level configuration where you might not want to confuse users with how it's implemented.

New in version 0.5.

Prepending the namespace

If you have a component that only gets used with one namespace, then it will probably help users if the documentation includes the full configuration key with the namespace prepended.

You can do that like this:

```
.. autocomponent:: collector.external.boto.crashstorage.BotoS3CrashStorage
   :namespace: crashstorage
```

Then the docs will show keys like `crashstorage_foo` rather than just `foo`.

New in version 0.8.

Showing keys as uppercased or lowercased

If your project primarily depends on configuration from OS environment variables, then you probably want to document those variables with the keys shown as uppercased.

You can do that like this:

```
.. autocomponent:: collector.external.boto.crashstorage.BotoS3CrashStorage
   :case: upper
```

If your project primarily depends on configuration from INI files, then you probably want to document those variables with keys shown as lowercased.

You can do that like this:

```
.. autocomponent:: collector.external.boto.crashstorage.BotoS3CrashStorage
   :case: lower
```

New in version 0.8.

5.5 Recipes

This contains some ways of solving problems I've had with applications I use Everett in. These use cases help me to shape the Everett architecture such that it's convenient and flexible, but not big and overbearing.

Hopefully they help you, too.

If there are things you're trying to solve and you're using Everett that aren't covered here, add an item to the issue tracker.

Contents

- *Recipes*
  - *Centralizing configuration specification*
  - *Using components that share configuration by passing arguments*
  - *Using components that share configuration using alternate keys*

5.5.1 Centralizing configuration specification

It's easy to set up a *everett.manager.ConfigManager* and then call it for configuration. However, with any non-trivial application, it's likely you're going to refer to configuration options multiple times in different parts of the code.

One way to do this is to pull out the configuration value and store it in a global constant or an attribute somewhere and pass that around.

Another way to do this is to create a configuration component, define all the configuration options there and then pass that component around.

For example, this creates an `AppConfig` component which has configuration for the application:

```
import logging

from everett.component import ConfigOptions, RequiredConfigMixin
from everett.manager import ConfigManager


def parse_loglevel(value):
    text_to_level = {
        'CRITICAL': 50,
        'ERROR': 40,
        'WARNING': 30,
        'INFO': 20,
        'DEBUG': 10
    }
    try:
        return text_to_level[value.upper()]
    except KeyError:
        raise ValueError(
            '"%s" is not a valid logging level. Try CRITICAL, ERROR, '
            'WARNING, INFO, DEBUG' % value
        )


class AppConfig(RequiredConfigMixin):
    required_config = ConfigOptions()
    required_config.add_option(
        'debug',
        parser=bool,
        default='false',
        doc='Turns on debug mode for the applciation'
    )
    required_config.add_option(
        'loglevel',
        parser=parse_loglevel,
        default='INFO',
        doc='Log level for the application'
    )

    def __init__(self, config):
        self.raw_config = config
        self.config = config.with_options(self)

    def __call__(self, *args, **kwargs):
        return self.config(*args, **kwargs)


def init_app():
    config = ConfigManager.from_dict({})
    app_config = AppConfig(config)

    logging.basicConfig(level=app_config('loglevel'))
```

(continues on next page)

(continued from previous page)

```
    if app_config('debug'):
        logging.info('debug mode!')


if __name__ == '__main__':
    init_app()
```

Couple of nice things here. First, is that if you do Sphinx documentation, you can use `autocomponent` to automatically document your configuration based on the code. Second, you can use *everett.component.RequiredConfigMixin.get_runtime_config()* to print out the runtime configuration at startup.

5.5.2 Using components that share configuration by passing arguments

Say we have multiple components that share some configuration value that's probably managed by another component.

For example, a "basedir" configuration value that defines the root directory for all the things this application does things with.

Let's create an app component which creates two file system components passing them a basedir:

```
import os

from everett.component import RequiredConfigMixin, ConfigOptions
from everett.manager import ConfigManager, parse_class


class App(RequiredConfigMixin):
    required_config = ConfigOptions()
    required_config.add_option(
        'basedir'
    )
    required_config.add_option(
        'reader',
        parser=parse_class
    )
    required_config.add_option(
        'writer',
        parser=parse_class
    )

    def __init__(self, config):
        self.config = config.with_options(self)

        self.basedir = self.config('basedir')
        self.reader = self.config('reader')(config, self.basedir)
        self.writer = self.config('writer')(config, self.basedir)


class FSReader(RequiredConfigMixin):
    required_config = ConfigOptions()
    required_config.add_option(
        'file_type',
        default='json'
    )
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, config, basedir):
        self.config = config.with_options(self)
        self.read_dir = os.path.join(basedir, 'read')


class FSWriter(RequiredConfigMixin):
    required_config = ConfigOptions()
    required_config.add_option(
        'file_type',
        default='json'
    )

    def __init__(self, config, basedir):
        self.config = config.with_options(self)
        self.write_dir = os.path.join(basedir, 'write')


config = ConfigManager.from_dict({
    'BASEDIR': '/tmp',
    'READER': '__main__.FSReader',
    'WRITER': '__main__.FSWriter',

    'READER_FILE_TYPE': 'json',

    'WRITER_FILE_TYPE': 'yaml'
})


app = App(config)
assert app.reader.read_dir == '/tmp/read'
assert app.writer.write_dir == '/tmp/write'
```

Why do it this way?

In this scenario, the `basedir` is defined at the app-scope and is passed to the reader and writer classes when they're created. In this way, `basedir` is app configuration, but not reader/writer configuration.

5.5.3 Using components that share configuration using alternate keys

Say we have two components that share a set of credentials. We don't want to have to specify the same set of credentials twice, so instead, we use alternate keys which let you specify other keys to look at for a configuration value. This lets us have both components look at the same keys for their credentials and then we only have to define them once.

Let's create a db reader and a db writer component:

```
from everett.component import RequiredConfigMixin, ConfigOptions
from everett.manager import ConfigManager


class DBReader(RequiredConfigMixin):
    required_config = ConfigOptions()
    required_config.add_option(
        'username',
        alternate_keys=['root:db_username']
    )
    required_config.add_option(
```

(continues on next page)

(continued from previous page)

```
        'password',
        alternate_keys=['root:db_password']
    )

    def __init__(self, config):
        self.config = config.with_options(self)


class DBWriter(RequiredConfigMixin):
    required_config = ConfigOptions()
    required_config.add_option(
        'username',
        alternate_keys=['root:db_username']
    )
    required_config.add_option(
        'password',
        alternate_keys=['root:db_password']
    )

    def __init__(self, config):
        self.config = config.with_options(self)


# Define a shared configuration
config = ConfigManager.from_dict({
    'DB_USERNAME': 'foo',
    'DB_PASSWORD': 'bar'
})

reader = DBReader(config.with_namespace('reader'))
assert reader.config('username') == 'foo'
assert reader.config('password') == 'bar'

writer = DBWriter(config.with_namespace('writer'))
assert writer.config('username') == 'foo'
assert writer.config('password') == 'bar'


# Or define different credentials
config = ConfigManager.from_dict({
    'READER_USERNAME': 'joe',
    'READER_PASSWORD': 'foo',

    'WRITER_USERNAME': 'pete',
    'WRITER_PASSWORD': 'bar',
})

reader = DBReader(config.with_namespace('reader'))
assert reader.config('username') == 'joe'
assert reader.config('password') == 'foo'

writer = DBWriter(config.with_namespace('writer'))
assert writer.config('username') == 'pete'
assert writer.config('password') == 'bar'
```

5.6 Library

5.6.1 everett.manager

This module contains the configuration infrastructure allowing for deriving configuration from specified sources in the order you specify.

class everett.manager.**BoundConfig**(*config*, *options*)

Wraps a config and binds it to a set of options

This restricts the config to only return keys from the option set. Further, it uses the option set to determine the default and the parser for that option.

This is useful for binding configuration to a component's specified options.

get_namespace()

Retrieves the complete namespace for this config object

Returns namespace as a list of strings

class everett.manager.**ConfigDictEnv**(*cfg*)

Source for pulling configuration out of a dict

This is handy for testing. You might also use it if you wanted to move all your defaults values into one centralized place.

Keys are prefixed by namespaces and the whole thing is uppercased.

For example, namespace "bar" for key "foo" becomes `BAR_FOO` in the dict.

For example:

```
from everett.manager import ConfigDictEnv, ConfigManager

config = ConfigManager([
    ConfigDictEnv({
        'FOO_BAR': 'someval',
        'BAT': '1',
    })
])
```

Keys are not case sensitive. This also works:

```
from everett.manager import ConfigDictEnv, ConfigManager

config = ConfigManager([
    ConfigDictEnv({
        'foo_bar': 'someval',
        'bat': '1',
    })
])

print config('foo_bar')
print config('FOO_BAR')
print config.with_namespace('foo')('bar')
```

Also, `ConfigManager` has a convenience classmethod for creating a `ConfigManager` with just a dict environment:

```
from everett.manager import ConfigManager

config = ConfigManager.from_dict({
    'FOO_BAR': 'bat'
})
```

Changed in version 0.3: Keys are no longer case-sensitive.

class everett.manager.**ConfigEnvFileEnv**(*possible_paths*)

Source for pulling configuration out of .env files

This source lets you specify configuration in an .env file. This is useful for local development when in production you use values in environment variables.

Keys are prefixed by namespaces and the whole thing is uppercased.

For example, key "foo" will be `FOO` in the file.

For example, namespace "bar" for key "foo" becomes `BAR_FOO` in the file.

Key and namespace can consist of alphanumeric characters and _.

To use, instantiate and toss in the source list:

```
from everett.manager import ConfigEnvFileEnv, ConfigManager

config = ConfigManager([
    ConfigEnvFileEnv('.env')
])
```

For multiple paths:

```
from everett.manager import ConfigEnvFileEnv, ConfigManager

config = ConfigManager([
    ConfigEnvFileEnv([
        '.env',
        'config/prod.env'
    ])
])
```

Here's an example .env file:

```
DEBUG=true

# secrets
SECRET_KEY=ou812

# database setup
DB_HOST=localhost
DB_PORT=5432
```

class everett.manager.**ConfigManager**(*environments*, *doc=''*, *with_override=True*)

Manages multiple configuration environment layers

classmethod basic_config(*env_file='.env'*)

Returns a basic ConfigManager

This sets up a ConfigManager that will look for configuration in this order:

1. environment

2. specified `env_file` defaulting to `.env`

This is for a fast one-line opinionated setup.

Example:

```
from everett.manager import ConfigManager

config = ConfigManager.basic_config()
```

This is shorthand for:

```
config = ConfigManager(
    environments=[
        ConfigOSEnv(),
        ConfigEnvFileEnv(['.env'])
    ]
)
```

Parameters **env_file** – the name of the env file to use

Returns a *everett.manager.ConfigManager*

classmethod from_dict(*dict_config*)

Creates a ConfigManager with specified configuration as a Python dict

This is shorthand for:

```
config = ConfigManager([ConfigDictEnv(dict_config)])
```

This is handy for writing tests for the app you're using Everett in.

Parameters **dict_config** – Python dict holding the configuration for this manager

Returns ConfigManager with specified configuration

New in version 0.3.

class everett.manager.**ConfigOSEnv**

Source for pulling configuration out of the environment

This source lets you specify configuration in the environment. This is useful for infrastructure related configuration like usernames and ports and secret configuration like passwords.

Keys are prefixed by namespaces and the whole thing is uppercased.

For example, key "foo" will be `FOO` in the environment.

For example, namespace "bar" for key "foo" becomes `BAR_FOO` in the environment.

Key and namespace can consist of alphanumeric characters and _.

Note: Unlike other config environments, this one is case sensitive in that keys defined in the environment **must** be all uppercase.

For example, these are good:

```
FOO=bar
FOO_BAR=bar
FOO_BAR1=bar
```

This is bad:

```
foo=bar
```

To use, instantiate and toss in the source list:

```
from everett.manager import ConfigOSEnv, ConfigManager

config = ConfigManager([
    ConfigOSEnv()
])
```

class everett.manager.**ConfigObjEnv**(*obj*, *force_lower=True*)

Source for pulling configuration values out of a Python object

This is handy for a few weird situations. For example, you can use this to "bridge" Everett configuration with command line arguments. The argparse Namespace works fine here.

Namespace (the Everett one–not the argparse one) is prefixed. So key "foo" in namespace "bar" is "foo_bar".

For example:

```
import argparse

from everett.manager import ConfigObjEnv, ConfigManager

parser = argparse.ArgumentParser()
parser.add_argument(
    '--debug', help='to debug or not to debug'
)
parsed_vals = parser.parse_known_args()[0]

config = ConfigManager([
    ConfigObjEnv(parsed_vals)
])

print config('debug', parser=bool)
```

Keys are not case-sensitive–everything is converted to lowercase before pulling it from the object.

Note: ConfigObjEnv has nothing to do with the library configobj.

New in version 0.6.

class everett.manager.**ConfigOverride**(***cfg*)

Allows you to override config for writing tests

This can be used as a class decorator:

```
@config_override(FOO='bar', BAZ='bat')
class FooTestClass(object):
    ...
```

This can be used as a function decorator:

```
@config_override(FOO='bar')
def test_foo():
    ...
```

This can also be used as a context manager:

```
def test_foo():
    with config_override(FOO='bar'):
        ...
```

pop_config()
Pops a config layer off

Raises **IndexError** – If there are no layers to pop off

push_config()
Pushes self._cfg as a config layer onto the stack

class everett.manager.**ConfigOverrideEnv**
Override configuration layer for testing

class everett.manager.**ListOf**(*parser*, *delimiter=','*)
Parses a comma-separated list of things

```
>>> ListOf(str)('')
[]
>>> ListOf(str)('a,b,c,d')
['a', 'b', 'c', 'd']
>>> ListOf(int)('1,2,3,4')
[1, 2, 3, 4]
```

Note: This doesn't handle quotes or backslashes or any complicated string parsing.

For example:

```
>>> ListOf(str)('"a,b",c,d')
['"a', 'b"', 'c', 'd']
```

class everett.manager.**NamespacedConfig**(*config*, *namespace*)
Applies a namespace to a config

This restricts keys in a config to those belonging to the specified namespace.

get_namespace()
Retrieves the complete namespace for this config object

Returns namespace as a list of strings

everett.manager.**config_override**
alias of *everett.manager.ConfigOverride*

everett.manager.**get_key_from_envs**(*envs*, *key*)
Return the value of a key from the given dict respecting namespaces.

Data can also be a list of data dicts.

everett.manager.**get_parser**(*parser*)
Returns a parsing function for a given parser

everett.manager.**listify**(*thing*)
Returns a new list of thing

If thing is a string, then returns a list of thing. Otherwise returns thing.

Parameters **thing** – string or list of things

Returns list

everett.manager.**parse_bool**(*val*)
Parses a bool

Handles a series of values, but you should probably standardize on "true" and "false".

```
>>> parse_bool('y')
True
>>> parse_bool('FALSE')
False
```

everett.manager.**parse_class**(*val*)
Parses a string, imports the module and returns the class

```
>>> parse_class('hashlib.md5')
<built-in function openssl_md5>
```

everett.manager.**parse_env_file**(*envfile*)
Parse the content of an iterable of lines as .env

Return a dict of config variables.

```
>>> parse_env_file(['DUDE=Abides'])
{'DUDE': 'Abides'}
```

everett.manager.**qualname**(*thing*)
Returns the dot name for a given thing

```
>>> import everett.manager
>>> qualname(str)
'str'
>>> qualname(everett.manager.parse_class)
'everett.manager.parse_class'
>>> qualname(everett.manager)
'everett.manager'
```

5.6.2 everett.component

Module holding infrastructure for building components.

class everett.component.**ConfigOptions**
Class for holding a collection of config options

add_option(*key*, *default=NO_VALUE*, *alternate_keys=NO_VALUE*, *doc=''*, *parser=<class 'str'>*)
Adds an option to the group

Parameters

- **key** – the key to look up
- **default** – the default value (if any); must be a string that is parseable by the specified parser
- **alternate_keys** – the list of alternate keys to look up; supports a `root:` key prefix which will cause this to look at the configuration root rather than the current namespace
- **doc** – documentation for this config option
- **parser** – the parser for converting this value to a Python object

class everett.component.**RequiredConfigMixin**

Mixin for component classes that have required configuration

As with all mixins, make sure this is earlier in the class list.

Example:

```
from everett.component import RequiredConfigMixin, ConfigOptions

class SomeComponent(RequiredConfigMixin):
    required_config = ConfigOptions()
    required_config.add_option('foo')

    def __init__(self, config):
        self.config = config.with_options(self)
```

classmethod get_required_config()

Rolls up the configuration for class and parent classes

Returns final `ConfigOptions` representing all configuration for this class

get_runtime_config(*namespace=None*)

Roll up the runtime config for this class and all children

Implement this to call `.get_runtime_config()` on child components or to adjust how it works.

For example, if you created a component that has a child component, you could do something like this:

```
class MyComponent(RequiredConfigMixin):
    ....

    def __init__(self, config):
        self.config = config.with_options(self)
        self.child = OtherComponent(config.with_namespace('source'))

    def get_runtime_config(self, namespace=None):
        for item in super(MyComponent, self).get_runtime_config(namespace):
            yield item
        for item in self.child.get_runtime_config(['source']):
            yield item
```

Calling this function can give you the complete runtime configuration for a component tree. This is helpful for doing things like printing the configuration being used including default values.

Note: If this instance has a `.config` attribute and it is a `everett.component.BoundConfig`, then this will try to compute the runtime config.

Otherwise, it'll yield nothing.

Parameters **namespace** (*list*) – list of namespace parts or None

Returns list of `(namespace, key, option)`

5.7 Hacking

5.7.1 Release process

1. Checkout master tip.
2. Check to make sure `setup.py` and requirements files have correct versions of requirements.
3. Update version numbers in `everett/__init__.py`.
    1. Set `__version__` to something like `1.0.0` (use semver).
    2. Set `__releasedate__` to something like `20190107`.
4. Update `HISTORY.rst`
    1. Set the date for the release.
    2. Make sure to note any backwards incompatible changes.
5. Verify correctness.
    1. Run tests.
    2. Build docs (this runs example code).
    3. Verify all that works.
6. Tag the release:

    ```
    $ git tag -a v1.0.0
    ```

    Copy the details from `HISTORY.rst` into the tag comment.
7. Update PyPI:

    ```
    $ rm -rf dist/*
    $ python setup.py sdist bdist_wheel
    $ twine upload dist/*
    ```

8. Push everything:

    ```
    $ git push --tags official master
    ```

9. Announce the release.

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

e

everett.component, 48
everett.manager, 43
everett.sphinxext, 36

Index

Symbols

__call__() (*everett.manager.ConfigManager method*), 27

A

add_option() (*everett.component.ConfigOptions method*), 48

B

basic_config() (*everett.manager.ConfigManager class method*), 44
BoundConfig (*class in everett.manager*), 43

C

config_override (*in module everett.manager*), 47
ConfigDictEnv (*class in everett.manager*), 43
ConfigEnvFileEnv (*class in everett.manager*), 44
ConfigManager (*class in everett.manager*), 44
ConfigObjEnv (*class in everett.manager*), 46
ConfigOptions (*class in everett.component*), 48
ConfigOSEnv (*class in everett.manager*), 45
ConfigOverride (*class in everett.manager*), 46
ConfigOverrideEnv (*class in everett.manager*), 47
ConfigurationError (*class in everett*), 29
ConfigurationMissingError (*class in everett*), 30

E

everett.component (*module*), 48
everett.manager (*module*), 43
everett.sphinxext (*module*), 36

F

from_dict() (*everett.manager.ConfigManager class method*), 45

G

get_key_from_envs() (*in module everett.manager*), 47
get_namespace() (*everett.manager.BoundConfig method*), 43
get_namespace() (*everett.manager.NamespacedConfig method*), 47
get_parser() (*in module everett.manager*), 47
get_required_config() (*everett.component.RequiredConfigMixin class method*), 49
get_runtime_config() (*everett.component.RequiredConfigMixin method*), 49

I

InvalidKeyError (*class in everett*), 30
InvalidValueError (*class in everett*), 29

L

listify() (*in module everett.manager*), 47
ListOf (*class in everett.manager*), 47

N

NamespacedConfig (*class in everett.manager*), 47

P

parse_bool() (*in module everett.manager*), 47
parse_class() (*in module everett.manager*), 48
parse_env_file() (*in module everett.manager*), 48
pop_config() (*everett.manager.ConfigOverride method*), 47
push_config() (*everett.manager.ConfigOverride method*), 47

Q

qualname() (*in module everett.manager*), 48

R

RequiredConfigMixin (*class in everett.component*), 48