Everett Documentation

Release 2.0.1 20210823

Will Kahn-Greene

Jan 13, 2022

CONTENTS

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

ONE

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, YAML 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

TWO

QUICK START

2.1 Fast start example

You're writing an app and want it to look for configuration:

1. in an `.env` file in the current working directory
2. then in the process environment

You set up a configuration manager like this:

```
from everett.manager import ConfigManager

config = ConfigManager.basic_config()
```

and use it to get configuration values like this:

```
api_host = config("api_host")
max_bytes = config("max_bytes", default="1000", parser=int)
debug_mode = config("debug", default="False", parser=bool)
```

You can create a basic configuration that points to your documentation like this:

```
config = ConfigManager.basic_config(
    doc="Check https://example.com/configuration for docs."
)
```

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

```
$ DEBUG=foo python myprogram.py
<traceback>
DEBUG requires a value parseable by bool
DEBUG docs: Set to True for debugmode; False for regular mode.
Project docs: Check https://example.com/configuration for docs.
```

You can use `config_override` in your tests to test various configuration values:

```
from everett.manager import config_override

from myapp import debug_mode


```

(continues on next page)

(continued from previous page)

```
def test_debug_on():
    with config_override(DEBUG="on"):
        assert debug_mode == True

def test_debug_off():
    with config_override(DEBUG="off"):
        assert debug_mode == False
```

When you outgrow that or need different variations of it, you can switch to creating a `ConfigManager` instance that meets your needs.

2.2 More complex example

You're writing an app and want to pull configuration (in order of prcedence):

1. from the process environment
2. from an INI file stored in a place specified by (in order of precedence):
    1. `MYAPP_INI` in the environment
    2. `~/.myapp.ini`
    3. `/etc/myapp.ini`

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

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

Then set up the `ConfigManager`:

```
# myapp.py

import os
import sys

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


CONFIG = 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(
            possible_paths=[
                os.environ.get("MYAPP_INI"),
                "~/.myapp.ini",
                "/etc/myapp.ini"
            ]
```

(continues on next page)

(continued from previous page)

```
        ),
    ],

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

Then use it:

```
# myapp.py continued

def is_debug(config):
    return config(
        "debug",
        default="False",
        parser=bool,
        doc="Set to True for debugmode; False for regular mode."
    )

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 CONFIG, is_debug

from everett.manager import config_override


@config_override(DEBUG="true")
def test_debug_true():
    assert is_debug(CONFIG) is True


def test_debug_false():
    with config_override(DEBUG="false"):
        assert is_debug(CONFIG) is False
```

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

```
$ DEBUG=foo python myprogram.py
<traceback>
DEBUG requires a value parseable by bool
DEBUG docs: Set to True for debugmode; False for regular mode.
Project docs: Check https://example.com/configuration for docs.
```

2.3 Configuration classes

Everett supports centralizing your configuration in a class. Instead of having configuration-related bits defined across your codebase, you can define it in a class. Let's rewrite the above example using a configuration class.

First, create a configuration class:

```
# myapp.py

import os
import sys

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


class AppConfig:
    class Config:
        debug = Option(
            parser=bool,
            default="false",
            doc="Switch debug mode on and off.")
        )
```

Then we set up a `ConfigManager` to look at the process environment for configuration and bound to the configuration options specified in `AppConfig`:

```
# myapp.py continued

def get_config():
    manager = ConfigManager(
        # Specify environments to check for configuration
        environments=[
            ConfigOSEnv(),
        ],

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

    # Apply the configuration class to the configuration manager
    # so that it handles option properties like defaults, parsers,
    # documentation, and so on.
    return manager.with_options(AppConfig())
```

Then use it:

```
# myapp.py continued

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 has some 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

2.4 Everett components

Everett supports components that require configuration. Say your app needs to connect to RabbitMQ. With Everett, you can define the component's configuration needs in the component class:

```
from everett.manager import Option


class RabbitMQComponent:
    class Config:
        host = Option(doc="RabbitMQ host to connect to")
        port = Option(default="5672", doc="Port to use", parser=int)
        queue_name = Option(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 you could instantiate a `RabbitMQComponent` that looks for configuration keys in the `rmq` namespace:

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

The `RabbitMQComponent` has a `HOST` key, so your configuration would need to define `RMQ_HOST`.

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
   :namespace: rmq
```

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

```
from everett.manager import ConfigManager

config = ConfigManager.basic_config()
```

(continues on next page)

(continued from previous page)

```
# Apply the "rmq" namespace to the configuration so all keys are
# prepended with RMQ_
rmq_config = config.with_namespace("rmq")

# Create a RabbitMQComponent with RMQ_REGULAR_ prepended to keys
regular_queue = RabbitMQComponent(rmq_config.with_namespace("regular"))

# Create a RabbitMQComponent with RMQ_PRIORITY_ prepended to keys
priority_queue = RabbitMQComponent(rmq_config.with_namespace("priority"))
```

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

THREE

INSTALL

3.1 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]'
```

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

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

3.2 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 -e '.[dev,ini,yaml]'
```

CHAPTER

FOUR

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

FIVE

CONTENTS

5.1 Configuration

Contents

        * *Implementing your own parsers*
    - *Trouble-shooting and logging what happened*

5.1.1 Setting up configuration in your app

Create a ConfigManager and specify sources

Configuration is handled by a `ConfigManager`. The `ConfigManager` handles looking up configuration keys across specified sources to resolve them to a value.

You can create a basic `ConfigManager` like this:

```
from everett.manager import ConfigManager


# basic_config() creates a ConfigManager that looks at a .env
# file and the process environment
config = ConfigManager.basic_config()

assert config("foo_var", default="bar") == "bar"
```

You can also specify the source environments in the order you want them looked at.

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"}),
    ]
)

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

Specify pointer to configuration documentation

In addition to a list of sources, you can provide a doc argument. You can use this to guide users hitting configuration errors to configuration documentation.

For example:

```
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 to 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>
    main()
  File "configuration_doc.py", line 12, in main
```

(continues on next page)

(continued from previous page)

```
    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.

5.1.2 Where to put ConfigManager instance

You can create the `ConfigManager` when instantiating an app class as a property on that instance–that works fine.

You could create the `ConfigManager` as a module-level singleton. That's fine, too.

`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.

5.1.3 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",
```

(continues on next page)

(continued from previous page)

```
        "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.

Parameters **cfg** (*Dict*) –

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
```

Parameters **possible_paths** (*Union[str, List[str]]*) –

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.

Parameters

- **obj** (*Any*) –
- **force_lower** (*bool*) –

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(possible_paths=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(
        possible_paths=[
            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(possible_paths="config/local.ini"),
    ConfigIniEnv(possible_paths="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]
```

(continues on next page)

(continued from previous page)

```
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

Parameters **possible_paths** (*Union[str, List[str]]*) – either a single string with a file path (e.g. "/etc/project.ini" or a list of strings with file paths

Return type None

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:
    namespace2:
        foo: "bar"
```

Giving you these namespaced keys:

- `FOO`
- `FOO2`
- `NAMESPACE_FOO`
- `NAMESPACE_NAMEPSACE2_FOO`

Parameters **possible_paths** (*Union[str, List[str]]*) – either a single string with a file path (e.g. `"/etc/project.yaml"` or a list of strings with file paths

Return type None

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):
```

(continues on next page)

(continued from previous page)

```
    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 *everett.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.

5.1.4 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=None*, *doc=''*, *parser=<class 'str'>*, *raise_error=True*, *raw_value=False*)

Return a parsed value from the environment.

Parameters

- **key** (*str*) – the key to look up
- **namespace** (*Optional[List[str]]*) – the namespace for the key–different environments use this differently
- **default** (*Union[str, everett.NoValue]*) – 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** (*Optional[List[str]]*) – 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** (*str*) – documentation for this config option

  New in version 0.6.

- **parser** (*Callable*) – the parser for converting this value to a Python object
- **raise_error** (*bool*) – True if you want a lack of value to raise a `everett.ConfigurationError`
- **raw_value** (*bool*) – 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
- *everett.InvalidValueError* – if the configuration value is invalid in some way (not an integer, not a bool, etc)

Return type Any

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 ListOf 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 your 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.

Note that the default value is always a string that's parseable by the parser.

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*.

If you're looking up values in the process environment, then the full key would be DB_USERNAME.

config("password", namespace="postgres", alternate_keys=["db_password", "root:postgres_password"]) 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**(*msg*, *namespace*, *key*, *parser*)
Error that indicates that the value is not valid.

Parameters

- **msg** (*str*) –
- **namespace** (*Optional[List[str]]*) –
- **key** (*str*) –
- **parser** (*Callable*) –

class everett.**ConfigurationMissingError**(*msg*, *namespace*, *key*, *parser*)
Error that indicates that required configuration is missing.

Parameters

- **msg** (*str*) –
- **namespace** (*Optional[List[str]]*) –

- **key** (*str*) –
- **parser** (*Callable*) –

class everett.**InvalidKeyError**
Error that indicates the key is not valid for this component.

5.1.5 Handling exceptions when extracting values

When the namespaced key isn't found in any of the sources, then Everett will always 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, doc="set debug mode")
except InvalidValueError:
    # The "debug_mode" configuration value is incorrect--alert
    # user in the logs.
    logging.exception("logged exception gah!")
```

That logs this:

```
ERROR:root:logged exception gah!
Traceback (most recent call last):
  File "/home/willkg/mozilla/everett/src/everett/manager.py", line 1197, in __call__
    parsed_val = parser(val)
  File "/home/willkg/mozilla/everett/src/everett/manager.py", line 226, in parse_bool
    raise ValueError(f"{val!r} is not a valid bool value")
ValueError: 'monkey' is not a valid bool value

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "code/configuration_handling_exceptions.py", line 13, in <module>
    some_val = config("debug_mode", parser=bool)
  File "/home/willkg/mozilla/everett/src/everett/manager.py", line 1217, in __call__
    raise InvalidValueError(msg, namespace, key, parser)
everett.InvalidValueError: ValueError: 'monkey' is not a valid bool value
DEBUG_MODE requires a value parseable by everett.manager.parse_bool
DEBUG_MODE docs: set debug mode
```

If that output won't be helpful to your users, you can catch the *everett.ConfigurationError* and log/print what will be helpful.

Also, you can change the structure of the error message by passing in a `msg_builder` argument to the *everett.manager.ConfigManager*.

For example, say your project is entirely done with INI configuration. Then you'd want to tailor the message accordingly.

```
import logging

from everett import InvalidValueError
from everett.manager import ConfigManager, ConfigDictEnv, qualname

logging.basicConfig()


def build_msg_for_ini(namespace, key, parser, msg="", option_doc="", config_doc=""):
    namespace = namespace or ["main"]
    namespace = "_".join(namespace)

    return (
        f"{key} in section [{namespace}] requires a value parseable by {qualname(parser)}
↪\n"
        + f"{key} in [{namespace}] docs: {option_doc}\n"
        + f"Project docs: {config_doc}"
    )


config = ConfigManager(
    environments=[ConfigDictEnv({"debug": "lizard"})],
    msg_builder=build_msg_for_ini,
)

try:
    debug_mode = config(
        "debug",
        default="false",
        parser=bool,
        doc="Set DEBUG=True to put the app in debug mode. Don't use this in production!",
    )
except InvalidValueError:
    logging.exception("logged exception gah!")
```

That logs this:

```
ERROR:root:logged exception gah!
Traceback (most recent call last):
  File "/home/willkg/mozilla/everett/src/everett/manager.py", line 1197, in __call__
    parsed_val = parser(val)
  File "/home/willkg/mozilla/everett/src/everett/manager.py", line 226, in parse_bool
    raise ValueError(f"{val!r} is not a valid bool value")
ValueError: 'lizard' is not a valid bool value

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "code/configuration_msg_builder.py", line 24, in <module>
```

(continues on next page)

(continued from previous page)

```
    debug_mode = config(
  File "/home/willkg/mozilla/everett/src/everett/manager.py", line 1217, in __call__
    raise InvalidValueError(msg, namespace, key, parser)
everett.InvalidValueError: debug in section [main] requires a value parseable by everett.
↪manager.parse_bool
debug in [main] docs: Set DEBUG=True to put the app in debug mode. Don't use this in␣
↪production!
Project docs:
```

5.1.6 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.

5.1.7 Parsers

All parsers are functions that take a string value and return a parsed instance.

For example:

- `int` takes a string value and returns an int.
- `parse_class` takes a dotted Python path string value and returns the class object
- `ListOf(str)` takes a string value and returns a list of strings

Note: When specifying configuration options, the default value must always be a string. When Everett can't find a value for a requested key, it will take the default value and pass it through the parser. Because parsers always take a string as input, the default value must always be a string.

This is valid:

```
debug = config("debug", parser=bool, default="false")
                                              ^^^^^^^
```

This is **not** valid:

```
debug = config("debug", parser=bool, default=False)
                                              ^^^^^
```

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`
- `decimal`
- `pathlib.Path`

bools

Everett provides a special bool parser that handles more descriptive 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*)

Parse a bool value.

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

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

Parameters **val** (*str*) –

Return type bool

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*)

Parse a string, imports the module and returns the class.

```
>>> parse_class("everett.manager.Option")
<class 'everett.manager.Option'>
```

Parameters **val** (*str*) –

Return type Any

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=','*)

Parse 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']
```

Parameters

- **parser** (*Callable*) –
- **delimiter** (*str*) –

dj_database_url

Everett works with dj-database-url. The `dj_database_url.parse` function takes a string and returns a Django database connection value.

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 depend on it or require it to be installed.

django-cache-url

Everett works 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

Implementing your own parser should be straight-forward. Parsing functions always take a string and return the Python value you need.

If the value is not parseable, the parsing function 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"})
```

(continues on next page)

(continued from previous page)

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

5.1.8 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.

5.2 Components

Contents

Changed in version 2.0: This is redone for v2.0.0 and simplified.

5.2.1 Building components with Everett

Everett allows you to build components that specify the configuration options for that component.

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, add a `Config` class to the class definition. In the `Config` class, specify the options for that class.

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

```
from everett.manager import Option

class RabbitMQComponent:
    class Config:
        host = Option(doc="RabbitMQ host to connect to")
        port = Option(default="5672", doc="Port to use", parser=int)
        queue_name = Option(doc="Queue to insert things into")

    def __init__(self, config):
```

(continues on next page)

(continued from previous page)

```
        # Bind the configuration manager to just the options
        # specified by this component. If this configuration manager
        # is asked to return a configuration option that's not
        # specified by # this class, then it'll throw an error.
        self.config = config.with_options(self)
        ...
```

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 could instantiate a `RabbitMQComponent` using an `rmq` configuration namespace like this:

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

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

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

```
rmq_regular = RabbitMQComponent(config.with_namespace("rmq_regular"))
rmq_priority = RabbitMQComponent(config.with_namespace("rmq_priority"))
```

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

Same component code–two different instances with two different configurations.

5.2.2 Subclassing

You can subclass components and override configuration options.

For example:

```
from everett.manager import ConfigManager, Option

class ComponentA:
    class Config:
        foo = Option(default="foo_from_a")
        bar = Option(default="bar_from_a")

class ComponentB(ComponentA):
    class Config:
        foo = Option(default="foo_from_b")

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

config = ConfigManager.basic_config()
compb = ComponentB(config)

print(compb.config("foo"))  # prints "foo_from_b"
print(compb.config("bar"))  # prints "bar_from_a"
```

5.2.3 Getting configuration information for components

You can get the configuration options for a component class using *everett.manager.get_config_for_class()*. This returns a dict of `configuration key -> (option, class)`. This helps with debugging which option came from which class.

everett.manager.**get_config_for_class**(*cls*)

Roll up configuration options for this class and parent classes.

This handles subclasses overriding configuration options in parent classes.

Parameters **cls** (*Type*) – the component class to return configuration options for

Returns final dict of configuration options for this class in `key -> (option, cls)` form

Return type Dict[str, Tuple[*everett.manager.Option*, Type]]

You can get the runtime configuration for a component or tree of components using *everett.manager.get_runtime_config()*. This returns a list of `(namespace, key, value, option, class)` tuples. The value is the computed runtime value taking into account the environments specified in the `ConfigManager` and class hierarchies.

It'll traverse any instance attributes that are components with options.

everett.manager.**get_runtime_config**(*config*, *component*, *traverse=<function traverse_tree>*)

Returns configuration specification and values for a component tree

For example, if you had a tree of components instantiated, you could traverse the tree and log the configuration:

```
from everett.manager import (
    ConfigManager,
    generate_uppercase_key,
    get_runtime_config,
    Option,
    parse_class,
)

class App:
    class Config:
        debug = Option(default="False", parser=bool)
        reader = Option(parser=parse_class)
        writer = Option(parser=parse_class)

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

        # App has a reader and a writer each of which has configuration
        # options
        self.reader = self.config("reader")(config.with_namespace("reader"))
        self.writer = self.config("writer")(config.with_namespace("writer"))

class Reader:
    class Config:
        input_file = Option()

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

(continues on next page)

(continued from previous page)

```
class Writer:
    class Config:
        output_file = Option()

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

cm = ConfigManager.from_dict(
    {
        # This specifies which reader component to use. Because we
        # specified this one, we need to define a READER_INPUT_FILE
        # value.
        "READER": "__main__.Reader",
        "READER_INPUT_FILE": "input.txt",

        # Same thing for the writer component.
        "WRITER": "__main__.Writer",
        "WRITER_OUTPUT_FILE": "output.txt",
    }
)

my_app = App(cm)

# This traverses the component tree starting with my_app and then
# traversing .reader and .writer attributes.
for namespace, key, value, option in get_runtime_config(cm, my_app):
    full_key = generate_uppercase_key(key, namespace)
    print(f"{full_key.upper()}={value or ''}")

# This should print out:
# DEBUG=False
# READER=__main__.Reader
# READER_INPUT_FILE=input.txt
# WRITER=__main__.Writer
# WRITER_OUTPUT_FILE=output.txt
```

Parameters

- **config** (everett.manager.ConfigManagerBase) – a configuration manager instance
- **component** (*Any*) – a component or tree of components
- **traverse** (*Callable*) – the function for traversing the component tree; see *everett.manager.traverse_tree()* for signature

Returns a list of (namespace, key, value, option) tuples

Return type List[Tuple[List[str], str, Any, *everett.manager.Option*]]

5.3 Documenting

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.

For example, with this code:

```
import logging

from everett.manager import ConfigManager, Option


TEXT_TO_LOGGING_LEVEL = {
    "CRITICAL": 50,
    "ERROR": 40,
    "WARNING": 30,
    "INFO": 20,
    "DEBUG": 10,
}


def parse_loglevel(value):
    try:
        return TEXT_TO_LOGGING_LEVEL[value.upper()]
    except KeyError:
        raise ValueError(
            f'"{value}" is not a valid logging level. Try CRITICAL, ERROR, '
            "WARNING, INFO, DEBUG"
        )


class AppConfig:
    class Config:
        debug = Option(
            parser=bool,
            default="false",
            doc="Turns on debug mode for the application",
        )
        loglevel = Option(
            parser=parse_loglevel,
            default="INFO",
            doc=(
                "Log level for the application; CRITICAL, ERROR, WARNING, INFO, "
                "DEBUG"
```

(continues on next page)

(continued from previous page)

```
            ),
        )


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

    logging.basicConfig(level=config("loglevel"))

    if config("debug"):
        logging.info("debug mode!")


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

You can add this to your docs:

```
.. autocomponent:: recipes_appconfig.AppConfig
   :hide-classname:
   :case: upper
```

And get this:

Configuration

Options

- **DEBUG** (bool) – Turns on debug mode for the application

  Defaults to 'false'.

- **LOGLEVEL** (recipes_appconfig.parse_loglevel) – Log level for the application; CRITICAL, ERROR, WARNING, INFO, DEBUG

  Defaults to 'INFO'.

Module docs:

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 reStructuredText 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.

5.4 Testing

Often you want to adjust configuration in your tests. Everett facilitates testing by providing an override mechanism.

everett.manager.**config_override**(***cfg*)

Allow 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"):
        ...
```

Parameters **cfg** (*str*) –

Return type *everett.manager.ConfigOverride*

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

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.manager import ConfigManager, Option


TEXT_TO_LOGGING_LEVEL = {
    "CRITICAL": 50,
    "ERROR": 40,
    "WARNING": 30,
    "INFO": 20,
    "DEBUG": 10,
}


def parse_loglevel(value):
    try:
        return TEXT_TO_LOGGING_LEVEL[value.upper()]
    except KeyError:
        raise ValueError(
            f'"{value}" is not a valid logging level. Try CRITICAL, ERROR, '
            "WARNING, INFO, DEBUG"
```

(continues on next page)

(continued from previous page)

```
        )


class AppConfig:
    class Config:
        debug = Option(
            parser=bool,
            default="false",
            doc="Turns on debug mode for the application",
        )
        loglevel = Option(
            parser=parse_loglevel,
            default="INFO",
            doc=(
                "Log level for the application; CRITICAL, ERROR, WARNING, INFO, "
                "DEBUG"
            ),
        )


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

    logging.basicConfig(level=config("loglevel"))

    if 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.manager.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.manager import ConfigManager, Option, parse_class


class App:
```

(continues on next page)

(continued from previous page)

```
    class Config:
        basedir = Option()
        reader = Option(parser=parse_class)
        writer = Option(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 FilesystemReader:
    class Config:
        file_type = Option(default="json")

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


class FilesystemWriter:
    class Config:
        file_type = Option(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__.FilesystemReader",
        "WRITER": "__main__.FilesystemWriter",
        "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.manager import ConfigManager, Option


class DatabaseReader:
    class Config:
        username = Option(alternate_keys=["root:db_username"])
        password = Option(alternate_keys=["root:db_password"])

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


class DatabaseWriter:
    class Config:
        username = Option(alternate_keys=["root:db_username"])
        password = Option(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 = DatabaseReader(config.with_namespace("reader"))
assert reader.config("username") == "foo"
assert reader.config("password") == "bar"

writer = DatabaseWriter(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 = DatabaseReader(config.with_namespace("reader"))
assert reader.config("username") == "joe"
assert reader.config("password") == "foo"
```

(continues on next page)

(continued from previous page)

```
writer = DatabaseWriter(config.with_namespace("writer"))
assert writer.config("username") == "pete"
assert writer.config("password") == "bar"
```

5.6 API

This is the API of functions and classes in Everett.

Configuration things:

- *everett.manager.ConfigManager*
- *everett.manager.Option*

Utility functions:

- *everett.manager.config_override*
- *everett.manager.generate_uppercase_key*
- *everett.manager.get_config_for_class*
- *everett.manager.get_runtime_config*
- *everett.manager.qualname*

Configuration environments:

- *everett.manager.ConfigObjEnv*
- *everett.manager.ConfigDictEnv*
- *everett.manager.ConfigEnvFileEnv*
- *everett.manager.ConfigOSEnv*
- (INI) *everett.ext.inifile.ConfigIniEnv*
- (YAML) *everett.ext.yamlfile.ConfigYamlEnv*

Errors:

- *everett.ConfigurationError*
- *everett.InvalidKeyError*
- *everett.ConfigurationMissingError*
- *everett.InvalidValueError*

Parsers:

- *everett.manager.parse_class()*
- *everett.manager.ListOf()*

5.6.1 everett

Everett is a Python library for configuration.

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

exception everett.**InvalidKeyError**
Error that indicates the key is not valid for this component.

exception everett.**DetailedConfigurationError**(*msg*, *namespace*, *key*, *parser*)
Base class for configuration errors that have a msg, namespace, key, and parser.

Parameters

- **msg** (*str*) –
- **namespace** (*Optional[List[str]]*) –
- **key** (*str*) –
- **parser** (*Callable*) –

exception everett.**ConfigurationMissingError**(*msg*, *namespace*, *key*, *parser*)
Error that indicates that required configuration is missing.

Parameters

- **msg** (*str*) –
- **namespace** (*Optional[List[str]]*) –
- **key** (*str*) –
- **parser** (*Callable*) –

exception everett.**InvalidValueError**(*msg*, *namespace*, *key*, *parser*)
Error that indicates that the value is not valid.

Parameters

- **msg** (*str*) –
- **namespace** (*Optional[List[str]]*) –
- **key** (*str*) –
- **parser** (*Callable*) –

5.6.2 everett.manager

Contains configuration infrastructure.

This module contains the configuration classes and functions for deriving configuration values from specified sources in the order specified.

everett.manager.**qualname**(*thing*)
Return the Python dotted name for a given thing.

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

(continues on next page)

(continued from previous page)

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

Parameters **thing** (*Any*) – the thing to get the qualname from

Returns the Python dotted name

Return type str

everett.manager.**build_msg**(*namespace*, *key*, *parser*, *msg=''*, *option_doc=''*, *config_doc=''*)

Builds a message for a configuration error exception

Parameters

- **namespace** (*Optional[List[str]]*) – list of strings or None that represent the configuration variable namespace
- **key** (*str*) – the configuration variable key
- **parser** (*Callable*) – the parser that will be used to parse the value for this configuration variable
- **msg** (*str*) – the error message
- **option_doc** (*str*) – the configuration option documentation
- **config_doc** (*str*) – the ConfigManager documentation

Returns the error message string

Return type str

class everett.manager.**Option**(*default=NO_VALUE*, *alternate_keys=None*, *doc=''*, *parser=<class 'str'>*, *meta=None*)

Settings for a single configuration option.

Parameters

- **default** (*Union[str, everett.NoValue]*) – 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** (*Optional[List[str]]*) – 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** (*str*) – documentation for this config option

  New in version 0.6.

- **parser** (*Callable*) – the parser for converting this value to a Python object
- **meta** (*Any*) – any meta information that's tied to this option; useful for noting which options are related in some way or which are secrets that shouldn't be logged

everett.manager.**get_config_for_class**(*cls*)

Roll up configuration options for this class and parent classes.

This handles subclasses overriding configuration options in parent classes.

Parameters **cls** (*Type*) – the component class to return configuration options for

Returns final dict of configuration options for this class in `key -> (option, cls)` form

Return type Dict[str, Tuple[*everett.manager.Option*, Type]]

everett.manager.**traverse_tree**(*instance*, *namespace=None*)

Traverses a tree of objects and computes the configuration for it

Note: This expects the tree not to have any loops or repeated nodes.

Parameters

- **instance** (*Any*) – the component to traverse
- **namespace** (*Optional[List[str]]*) – the list of strings forming the namespace or None

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

Return type Iterable[Tuple[List[str], str, *everett.manager.Option*, Any]]

everett.manager.**parse_bool**(*val*)

Parse a bool value.

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

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

Parameters **val** (*str*) –

Return type bool

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'}
```

Parameters **envfile** (*Iterable[str]*) –

Return type Dict

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

Parse a string, imports the module and returns the class.

```
>>> parse_class("everett.manager.Option")
<class 'everett.manager.Option'>
```

Parameters **val** (*str*) –

Return type Any

everett.manager.**get_parser**(*parser*)

Return a parsing function for a given parser.

Parameters **parser** (*Callable*) –

Return type Callable

everett.manager.**listify**(*thing*)

Convert thing to a list.

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

Parameters **thing** (*Any*) – string or list of things

Returns list

Return type List[Any]

everett.manager.**generate_uppercase_key**(*key*, *namespace=None*)

Given a key and a namespace, generates a final uppercase key.

```
>>> generate_uppercase_key("foo")
'FOO'
>>> generate_uppercase_key("foo", ["namespace"])
'NAMESPACE_FOO'
>>> generate_uppercase_key("foo", ["namespace", "subnamespace"])
'NAMESPACE_SUBNAMESPACE_FOO'
```

Parameters

- **key** (*str*) –
- **namespace** (*Optional[List[str]]*) –

Return type str

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.

Parameters

- **envs** (*Iterable[Any]*) –
- **key** (*str*) –

Return type Union[str, everett.NoValue]

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

Parse 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']
```

Parameters

- **parser** (*Callable*) –

- **delimiter** (*str*) –

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

get(*key*, *namespace=None*)
Retrieve value for key.

Parameters

- **key** (*str*) –
- **namespace** (*Optional[List[str]]*) –

Return type Union[str, everett.NoValue]

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.

Parameters

- **obj** (*Any*) –
- **force_lower** (*bool*) –

get(*key*, *namespace=None*)
Retrieve value for key.

Parameters

- **key** (*str*) –

- **namespace** (*Optional[List[str]]*) –

Return type Union[str, everett.NoValue]

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.

Parameters **cfg** (*Dict*) –

get(*key*, *namespace=None*)

Retrieve value for key.

Parameters

- **key** (*str*) –

- **namespace** (*Optional[List[str]]*) –

Return type Union[str, everett.NoValue]

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
```

Parameters **possible_paths** (*Union[str, List[str]]*) –

get(*key*, *namespace=None*)

Retrieve value for key.

Parameters

- **key** (*str*) –
- **namespace** (*Optional[List[str]]*) –

Return type Union[str, everett.NoValue]

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()
])
```

get(*key*, *namespace=None*)

Retrieve value for key.

Parameters

- **key** (*str*) –
- **namespace** (*Optional[List[str]]*) –

Return type Union[str, everett.NoValue]

class everett.manager.**ConfigManagerBase**

Base configuration manager class.

get_namespace()

Retrieve the complete namespace for this config object.

Returns namespace as a list of strings

Return type List[str]

with_options(*component*)

Apply options component options to this configuration.

Parameters **component** (*Any*) –

Return type *everett.manager.ConfigManagerBase*

with_namespace(*namespace*)
Apply namespace to this configuration.

Parameters **namespace** (*str*) –

Return type *everett.manager.ConfigManagerBase*

everett.manager.**get_runtime_config**(*config*, *component*, *traverse=<function traverse_tree>*)
Returns configuration specification and values for a component tree

For example, if you had a tree of components instantiated, you could traverse the tree and log the configuration:

```
from everett.manager import (
    ConfigManager,
    generate_uppercase_key,
    get_runtime_config,
    Option,
    parse_class,
)

class App:
    class Config:
        debug = Option(default="False", parser=bool)
        reader = Option(parser=parse_class)
        writer = Option(parser=parse_class)

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

        # App has a reader and a writer each of which has configuration
        # options
        self.reader = self.config("reader")(config.with_namespace("reader"))
        self.writer = self.config("writer")(config.with_namespace("writer"))

class Reader:
    class Config:
        input_file = Option()

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

class Writer:
    class Config:
        output_file = Option()

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

cm = ConfigManager.from_dict(
    {
        # This specifies which reader component to use. Because we
        # specified this one, we need to define a READER_INPUT_FILE
        # value.
        "READER": "__main__.Reader",
        "READER_INPUT_FILE": "input.txt",
```

(continues on next page)

(continued from previous page)

```
        # Same thing for the writer component.
        "WRITER": "__main__.Writer",
        "WRITER_OUTPUT_FILE": "output.txt",
    }
)

my_app = App(cm)

# This traverses the component tree starting with my_app and then
# traversing .reader and .writer attributes.
for namespace, key, value, option in get_runtime_config(cm, my_app):
    full_key = generate_uppercase_key(key, namespace)
    print(f"{full_key.upper()}={value or ''}")

# This should print out:
# DEBUG=False
# READER=__main__.Reader
# READER_INPUT_FILE=input.txt
# WRITER=__main__.Writer
# WRITER_OUTPUT_FILE=output.txt
```

Parameters

- **config** (everett.manager.ConfigManagerBase) – a configuration manager instance
- **component** (*Any*) – a component or tree of components
- **traverse** (*Callable*) – the function for traversing the component tree; see *everett.manager.traverse_tree()* for signature

Returns a list of (namespace, key, value, option) tuples

Return type List[Tuple[List[str], str, Any, *everett.manager.Option*]]

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

Wrap 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.

Parameters

- **config** (everett.manager.ConfigManagerBase) –
- **component_name** (*str*) –
- **options** (*Mapping[str, Any]*) –

get_namespace()

Retrieve the complete namespace for this config object.

Returns namespace as a list of strings

Return type List[str]

class everett.manager.**NamespacedConfig**(*config*, *namespace*)

Apply a namespace to a config.

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

Parameters

- **config** (everett.manager.ConfigManagerBase) –
- **namespace** (*str*) –

get_namespace()

Retrieve the complete namespace for this config object.

Returns namespace as a list of strings

Return type List[str]

class everett.manager.**ConfigManager**(*environments*, *doc=''*, *msg_builder=<function build_msg>*, *with_override=True*)

Manage multiple configuration environment layers.

Instantiate a ConfigManager.

Parameters

- **environments** (*List[Any]*) – list of configuration sources to look through in the order they should be looked through
- **doc** (*str*) – help text printed to users when they encounter configuration errors

  New in version 0.6.
- **msg_builder** (*Callable*) – function that takes arguments and builds an exception message intended to be printed or conveyed to the user

  For example:

```
def build_msg(namespace, key, parser, msg="", option_doc="", config_
↪doc=""):
    full_key = namespace or []
    full_key = "_".join(full_key + [key]).upper()

    return (
        f"{full_key} requires a value parseable by {qualname(parser)}
↪\n"
        + option_doc + "\n"
        + config_doc + "\n"
    )
```

- **with_override** (*bool*) – whether or not to insert the special override environment used for testing as the first environment in the list of sources

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

Return 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** (*str*) – the name of the env file to use
- **doc** (*str*) – help text printed to users when they encounter configuration errors

Returns a *everett.manager.ConfigManager*

Return type *everett.manager.ConfigManager*

classmethod from_dict(*dict_config*)

Create 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** (*Dict*) – Python dict holding the configuration for this manager

Returns ConfigManager with specified configuration

Return type *everett.manager.ConfigManager*

New in version 0.3.

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

Handle contexts and decoration for overriding config in testing.

Parameters **cfg** (*str*) –

push_config()

Push `self._cfg` as a config layer onto the stack.

Return type None

pop_config()

Pop a config layer off.

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

Return type None

decorate(*fun*)

Decorate a function for overriding configuration.

Parameters **fun** (*Callable*) –

Return type Callable

everett.manager.**config_override**(***cfg*)

Allow 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"):
        ...
```

Parameters **cfg** (*str*) –

Return type *everett.manager.ConfigOverride*

5.6.3 everett.ext.inifile

Holds the ConfigIniEnv environment.

To use this, you must install the optional requirements:

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

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(possible_paths=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(
        possible_paths=[
            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(possible_paths="config/local.ini"),
    ConfigIniEnv(possible_paths="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

Parameters **possible_paths** (*Union[str, List[str]]*) – either a single string with a file path (e.g. "/etc/project.ini" or a list of strings with file paths

Return type None

parse_ini_file(*path*)
Parse ini file at path and return dict.

Parameters **path** (*str*) –

Return type Dict

get(*key*, *namespace=None*)
Retrieve value for key.

Parameters

- **key** (*str*) –
- **namespace** (*Optional[List[str]]*) –

Return type Union[str, everett.NoValue]

5.6.4 everett.ext.yamlfile

Holds the ConfigYamlEnv environment.

To use this, you must install the optional requirements:

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

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:
    namespace2:
        foo: "bar"
```

Giving you these namespaced keys:

- `FOO`
- `FOO2`
- `NAMESPACE_FOO`
- `NAMESPACE_NAMEPSACE2_FOO`

Parameters **possible_paths** (*Union[str, List[str]]*) – either a single string with a file path (e.g. `"/etc/project.yaml"` or a list of strings with file paths

Return type None

parse_yaml_file(*path*)

Parse yaml file at `path` and return a dict.

Parameters **path** (*str*) –

Return type Dict

get(*key*, *namespace=None*)
Retrieve value for key.

Parameters

- **key** (*str*) –
- **namespace** (*Optional[List[str]]*) –

Return type Union[str, everett.NoValue]

5.7 History

5.7.1 2.0.1 (August, 23rd, 2021)

Fixes:

- Fix Sphinx warning about roles in Everett sphinxext. (#165)
- Fix `get_runtime_config` to work with slots (#166)

5.7.2 2.0.0 (July 27th, 2021)

Backwards incompatible changes:

- This radically reduces the boilerplate required to define components. It also improves the connections between things so it's easier to:
  - determine the configuration required for a single component (taking into account superclasses, overriding, etc)
  - determine the runtime configuration for a component tree given a configuration manager

  Previously, components needed to subclass RequiredConfigMixin and provide a "required_config" class attribute. Something like this:

```
from everett.component import RequiredConfigMixin, ConfigOptions

class SomeClass(RequiredConfigMixin):
    required_config = ConfigOptions()
    required_config.add_option(
        "some_option",
        default="42",
    )
```

  That's been slimmed down and now looks like this:

```
from everett.manager import Option

class SomeClass:
    class Config:
        some_option = Option(default="42")
```

That's much simpler and the underlying implementation code is less tangled and complex, too.

If you used `everett.component.RequiredConfigMixin` or `everett.component.ConfigOptions`, you'll need to update your classes.

If you didn't use those things, then you don't have to make any changes.

See the documentation on components for how it all works now.

- Changed the way configuration variables are referred to in configuration error messages. Previously, I tried to use a general way "namespace=something key=somethingelse" but that's confusing and won't match up with project documentation.

  I changed it to the convention used in the process environment and env files. For example, `FOO_BAR`.

  If you use INI or YAML for configuration, you can specify a `msg_builder` argument when you build the `ConfigManager` and build error messages tailored to your users.

Fixes:

- Switch to `src/` repository layout.
- Added type annotations and type checking during CI. (#155)
- Standardized on f-strings across the codebase.
- Switched Sphinx theme.
- Update of documentation, fleshed out and simplified examples, cleaned up language, reworked structure of API section (previously called Library or some unhelpful thing like that), etc.

5.7.3 1.0.3 (October 28th, 2020)

Backwards incompatible changes:

- Dropped support for Python 3.4. (#96)
- Dropped support for Python 3.5. (#116)

Fixes:

- Add support for Python 3.7. (#68)
- Add support for Python 3.8. (#102)
- Add support for Python 3.9. (#117)
- Reformatted code with Black, added Makefile, switched to GitHub Actions.
- Fix `get_runtime_config()` to infer namespaces. (#118)
- Fix `RemovedInSphinx50Warning`. (#115)
- Documentation fixes and clarifications.

5.7.4 1.0.2 (February 22nd, 2019)

Fixes:

- Improve documentation.
- Fix problems when there are nested `BoundConfigs`. Now they work correctly. (#90)
- Add "meta" to options letting you declare additional data on the option when you're adding it.

  For example, this lets you do things like mark options as "secrets" so that you know which ones to `******` out when logging your configuration. (#88)

5.7.5 1.0.1 (January 8th, 2019)

Fixes:

- Fix documentation issues.
- Package missing `everett.ext`. Thank you, dsblank! (#84)

5.7.6 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)
- Everett now has a YAML configuration environment. In order to use it, you need to install its requirements:

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

  Then you can import it like this:

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

  (#72)

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.

5.7.7 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.

5.7.8 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.

5.7.9 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.

5.7.10 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.

5.7.11 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.

5.7.12 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.

5.7.13 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.

5.7.14 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.

5.7.15 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.

5.7.16 0.1 (August 1st, 2016)

Initial writing.

5.8 Hacking

5.8.1 Release process

1. Checkout main tip.
2. Check to make sure `setup.py` and requirements files have correct versions of requirements.

   Check dev dependencies using `make checkrot`.
3. Update version numbers in `src/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. Check the manifest: `check-manifest`
   2. Run tests: `make test`
   3. Build docs (this runs example code): `make docs`
6. Tag the release:

   ```
   $ git tag --sign 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 origin main
   ```
9. Announce the release.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

e

INDEX

Symbols

B

C

D

E

F

G

I

L

M

N

O

P

Q

T

W