omniconf Documentation

Release 1.3.1

Nick Douma

Nov 12, 2019

Contents

Contents:

CHAPTER 1

Design principles

The design of omniconf is based around the following principles:

- Defining settings must be easy.
- Configuration of values must be easy.
- Multiple sources for configuration must be allowed and supported.
- Fine-grained configuration should be an option.
- Backends should be easy to implement.

Configuring an application can be hard, and it gets more complex if more than one way to configure must be supported. omniconf aims to separate definition of Settings and the loading of the Config, so that multiple Backends can be easily used and changed.

CHAPTER 2

Keys

All Settings and Configs are defined using a simple key. The key should only contain ASCII characters (altough this is not validated). The following are valid keys:

```
username
password
application.module.setting
```

Dots denote a section, and are mainly used to group similar keys. They can also be used by backends, the *ConfigObjBackend* backend for instance uses the dots to lookup keys in nested sections.

CHAPTER 3

Terminology

Setting A definition of a key, along with some metadata, like a type or default value.

Config A Setting that has been configured, by specifying value.

Key A Setting defines a key, which can later be used to set a Config value. A key is defined as a simple ascii only string. A key may contain dots, which are interpreted a sections. *app.database.username* is a typical example.

Backend A source of Config values. Also see *Supported backends*.

prefix Some backends may allow a prefix to be configured. `EnvBackend` for example prepends this to the environment it tries to read.

CHAPTER 4

Supported backends

The following backends are supported as of version 1.3.1:

4.1 backend interface

All backends implement the same interface, which allows for easy addition of new (or external backends).

class omniconf.backends.generic.**ConfigBackend**(*conf=None*)

Defines a configuration backend, which provides configuration values based on keys.

classmethod autoconfigure(*conf*, *autoconfigure_prefix*)

Called with a `ConfigRegistry`, the result of this method must be either a new instance of this class, or `None`. This method is automatically called during the autoconfigure phase.

classmethod autodetect_settings(*autoconfigure_prefix*)

Returns a tuple of `Setting` objects, that are required for *`autoconfigure()`* to complete successfully.

get_value(*setting*)

Retrieves the value for the given `Setting`.

get_values(*settings*)
Retrieves a list of `Setting`s all at once. Values are returned as a list of tuples containing the :class:.Setting‘ and value.

4.2 commandline arguments

Command line arguments are implemented using `argparse`. This backend is enabled by default.

class omniconf.backends.argparse.**ArgparseBackend**(*conf=None*, *prefix=None*)
Uses the current process arguments, and allows values in it to be retrieved using dotted keys with a specific prefix. By default no prefix is assumed.

classmethod autoconfigure(*conf*, *autoconfigure_prefix*)
Called with a `ConfigRegistry`, the result of this method must be either a new instance of this class, or `None`. This method is automatically called during the autoconfigure phase.

get_value(*setting*)
Retrieves the value for the given `Setting`.

get_values(*settings*)
Process the given list `Setting` objects, and retrieve the values. Keys are converted as follows:

- Dots are replaced by dashes (-).
- The key is lowercased.
- A prefix is attached to the key, if specified

This means that a key like section.value will be queried like `--prefix-section-value`. When no prefix is specified, `--section-value` is queried instead.

Special handling is added for boolean Settings with a default specified, which works as follows:

- Settings with *_type=bool* and no default will be processed as normal.
- Settings with *_type=bool*, and where the default value is True will be specified as an argparse argument with *action=store_false*.
- Settings with *_type=bool*, and where the default value is False will be specified as an argparse argument with *action=store_true*.

4.3 environment variables

Environments are read from `os.environ`. This backend is enabled by default.

class omniconf.backends.env.**EnvBackend**(*conf=None*, *prefix=None*)
Uses the current process Environment, and allows values in it to be retrieved using dotted keys with a specific prefix. By default no prefix is assumed.

classmethod autoconfigure(*conf*, *autoconfigure_prefix*)
Called with a `ConfigRegistry`, the result of this method must be either a new instance of this class, or `None`. This method is automatically called during the autoconfigure phase.

get_value(*setting*)
Retrieves the value for the given `Setting`. Keys are converted as follows:

- Dots are replaced by underscores
- The key is uppercased.

- A prefix is attached to the key

This means that a key like section.value will be queried like `PREFIX_SECTION_VALUE`. When no prefix is specified, `SECTION_VALUE` is queried instead.

4.4 ConfigObj files

Files in ConfigObj format are supported. This backend is only enabled if *omniconf.configobj.filename* is specified during setup.

class omniconf.backends.configobj.**ConfigObjBackend**(*conf*)

Uses a ConfigObj file (or `StringIO` instance) as a backend, and allows values in it to be retrieved using dotted keys.

Dots in the keys denote a section in the ConfigObj document. For instance, the key *section.subsection.key* will correspond to this document:

```
[section]
[[subsection]]
key=value
```

classmethod autoconfigure(*conf*, *autoconfigure_prefix*)

Called with a `ConfigRegistry`, the result of this method must be either a new instance of this class, or `None`. This method is automatically called during the autoconfigure phase.

classmethod autodetect_settings(*autoconfigure_prefix*)

Returns a tuple of `Setting` objects, that are required for *`autoconfigure()`* to complete successfully.

4.5 JSON files

Files in JSON format are supported. This backend is only enabled if *omniconf.json.filename* is specified during setup.

class omniconf.backends.json.**JsonBackend**(*conf*)

Uses a JSON string as a backend, and allows values in it to be retrieved using dotted keys.

classmethod autoconfigure(*conf*, *autoconfigure_prefix*)

Called with a `ConfigRegistry`, the result of this method must be either a new instance of this class, or `None`. This method is automatically called during the autoconfigure phase.

classmethod autodetect_settings(*autoconfigure_prefix*)

Returns a tuple of `Setting` objects, that are required for *`autoconfigure()`* to complete successfully.

4.6 YAML files

Files in YAML format are supported. This backend is only enabled if *omniconf.yaml.filename* is specified during setup. All YAML documents in the file are consumed.

class omniconf.backends.yaml.**YamlBackend**(*conf*)

Uses a YAML string as a backend, and allows values in it to be retrieved using dotted keys.

classmethod autoconfigure(*conf*, *autoconfigure_prefix*)

Called with a `ConfigRegistry`, the result of this method must be either a new instance of this class, or `None`. This method is automatically called during the autoconfigure phase.

classmethod autodetect_settings(*autoconfigure_prefix*)
Returns a tuple of `Setting` objects, that are required for *autoconfigure()* to complete successfully.

4.7 Hashicorp Vault

Hashicorp's Vault is supported by using its API. This backend requires several configuration keys to be defined during setup, see the documentation below for details.

class omniconf.backends.vault.**VaultBackend**(*conf=None*, *prefix=None*, *url=None*, *auth=None*, *credentials=None*, *base_path=None*)
Uses Hashicorp's Vault as a backend, and allows values in it to be retrieved using dotted keys.

Key translation

Dotted keys are translated into an URL path, which is then optionally prepended by the configured backend prefix. The last part of the path is used as a property to retrieve. If a base_path is also configured, it overrides the backend prefix.

For instance, a setting with key *setting.foo.bar* will be translated into path `setting/foo`, from which the property with key `bar` will be retrieved. Because Vault nodes are grouped by backend, it usually makes sense to define *base_path* as *secret*, which corresponds to the Generic backend of Vault. In this example, the example key will be translated into path `secret/setting/foo`, from which the property with key *bar* will be retrieved.

API Connection

The URL endpoint which omniconf will default to `http://localhost:8200`, and can be configured using the configuration key `omniconf.vault.url`, assuming the `autoconfigure_prefix` is set to *omniconf*.

Authentication

Vault's API requires some form of authentication, of which the following are supported:

- Tokens
- TLS certificates
- Username & Password
- LDAP
- App ID
- AppRole

Retrieval of Vault data requires an ACL to be defined, which goes beyond the scope of this documentation. omniconf only needs read rights on the keys it tries to access.

Selection of what authentication method is used depends on which configuration is present during setup. For all the following examples, the `autoconfigure_prefix` is assumed to be *omniconf*:

- Token authentication is used if `omniconf.vault.auth.token` is defined.
- TLS certificates authentication is used if both `omniconf.vault.auth.tls.cert.filename` and `omniconf.vault.auth.tls.key.filename` are defined.
- Username and Password authentication is used if both `omniconf.vault.auth.userpass.username` and `omniconf.vault.auth.userpass.password` are defined.
- LDAP authentication is used if both `omniconf.vault.auth.ldap.username` and `omniconf.vault.auth.ldap.password` are defined.

- App ID authentication is used if both `omniconf.vault.auth.appid.app_id` and `omniconf.vault.auth.appid.user_id` are defined.
- AppRole authentication is used if both `omniconf.vault.auth.approle.role_id` and `omniconf.vault.auth.approle.secret_id` are defined.

The above order is also the order in which the configuration values are looked up. The first one to satisfy the conditions is used, and no further attepts are made if configuration fails.

classmethod autoconfigure(*conf*, *autoconfigure_prefix*)
Called with a `ConfigRegistry`, the result of this method must be either a new instance of this class, or `None`. This method is automatically called during the autoconfigure phase.

classmethod autodetect_settings(*autoconfigure_prefix*)
Returns a tuple of `Setting` objects, that are required for *`autoconfigure()`* to complete successfully.

get_value(*setting*)
Retrieves the value for the given `Setting`.

CHAPTER 5

Setting types

When a `Setting` is defined, a type is also declared. By default, the value of a *Setting* is `str`, but any class or function that accepts a single parameter and returns a class instance can be used. The class or function passed to *_type* will be called with the value to process as its only parameter.

5.1 Built-in interpretation

Special cases are added to support `dict`, `list`, `tuple()` and `bool`, which are processed by `ast`. The implementation can be found in the `unrepr` method in `omniconf.config`:

`omniconf.config.`**`unrepr`**(*src*, *_type*)

Returns an interpreted value based on `src`. If `source` is already an instance of `_type`, no interpretation is performed.

This means that a *Setting* declared as such:

```
from omniconf import setting
setting("items", _type=list)
```

Which is provided by a backend with the following string:

```
"['foo', 'bar', 'baz']"
```

Will return a list that looks like this:

```
from omniconf import config
print(config("items"))
# ['foo', 'bar', 'baz']
```

For detailed information, see the `ast` documentation.

5.2 Custom interpretation and types

The most simple custom type looks like this:

```
def custom_type(src):
   return src
```

This example simply takes the input as provided, and returns it as-is. Custom types are not limited to functions, classes can also be used. Any class that has exactly one (mandatory) parameter is valid):

```
class CustomType(object):
   def __init__(self, src, foo=bar):
      self.src = src
```

Some custom types are provided with omniconf, which may be used as-is, but also serve as examples.

5.2.1 Enum

omniconf.types.**enum**(*values*)
Returns the original value if it is present in values, otherwise raises a RuntimeError.

```
enum_func = enum(["foo", "bar"])
print enum_func("foo")
# "foo"
print enum_func("baz")
# ...
# RuntimeError: Invalid value specified, must be one of: foo, bar
```

5.2.2 Separator Sequence

A somewhat fancy name for what one might normally call a comma separated list. The implementation is not limited to just commas however, and can use any string.

omniconf.types.**separator_sequence**(*separator*)
Returns a function that parses a string value, separates it into parts and stores it as a read-only sequence:

```
parser = separator_sequence(",")
print parser("a,b,c")
# ['a', 'b', 'c']
```

If the input value is already a sequence (but not a string), the value is returned as is. The sequence is an instance of *SeparatorSequence*, and can be used as one would normally use a (read-only) tuple or list.

class omniconf.types.**SeparatorSequence**(*string*, *separator*)
Splits the given string using the given separator, and provides a the result with a read-only Sequence interface.

5.2.3 String Boolean

omniconf.types.**string_bool**(*value*)
Returns False if the value is Falsish or "False", True if value is "True", or the original value otherwise.

5.2.4 String or False

omniconf.types.**string_or_false**(*value*)
Returns the given value as-is, unless the values equals "False". In that case, boolean False is returned.

CHAPTER 6

Usage

6.1 Basic usage

The most basic usage of omniconf requires the use of the `setting()`, `config()` and `omniconf_load()` functions:

`omniconf.`**`setting`**(*key*, *_type=<class 'str'>*, *required=False*, *default=None*, *help=None*, *registry=None*)
Register a new `Setting` with the given key.

`omniconf.`**`config`**(*key*, *registry=None*)
Retrieves the configured value for a given key. If no specific registry is specified, the value will be retrieved from the default `ConfigRegistry`.

`omniconf.`**`omniconf_load`**(*config_registry=None*, *backends=None*, *autoconfigure_prefix=None*)
Fill the provided `ConfigRegistry`, by default using all available backends (as determined by `autoconfigure_backends()`. If no `ConfigRegistry` is provided, the default `ConfigRegistry` is used. If unset, autoconfigure_prefix will default to "omniconf".

Define Settings using `setting()`:

```
from omniconf import setting
setting("app.username")
setting("app.hostname")
```

After defining the Settings, use `omniconf_load()` to load values:

```
from omniconf import omniconf_load
omniconf_load()
```

Afterwards, you can use `config()` to retrieve values.

```
>>> from omniconf import config
>>> print config("app.username")
"user"
```

By default, all Settings defined using `setting()` will be stored as `str`. To use another class, do this:

```
from omniconf import setting
setting("app.firstname", _type=unicode)
setting("app.load_order", _type=list)
```

Any class can be used. See *Setting types* for more information.

6.2 Advanced usage

By default all Settings and Configs are registered in global Registries. These are defined in their respective modules:

`omniconf.config.`**`DEFAULT_REGISTRY = <omniconf.config.ConfigRegistry object>`**
Global `ConfigRegistry` which will be used when no specific `ConfigRegistry` is defined.

`omniconf.setting.`**`DEFAULT_REGISTRY = <omniconf.setting.SettingRegistry object>`**
Global `SettingRegistry` which will be used when no specific `SettingRegistry` is defined.

This allows you to easily define Settings. Sometimes you might want to have specific Settings and Configs however. You can achieve this by specifying your own Registries:

```
from omniconf.setting import SettingRegistry
from omniconf.config import ConfigRegistry
from omniconf import omniconf_load

settings = SettingRegistry()
configs = ConfigRegistry(setting_registry=settings)

setting("app.username", registry=settings)

omniconf_load(config_registry=configs)
```

omniconf actually uses this mechanism to build the context needed for autoconfiguring. You can check this out in `autoconfigure_backends()`

`omniconf.loader.`**`autoconfigure_backends`**`(`*`autoconfigure_prefix=None`*`)`
Determine available backends, based on the current configuration available in the environment and command line. Backends can define a `Setting` that is required for proper autodetection.

The result of this function is a list of backends, that are configured and ready to use.

6.3 Autoconfigure prefix usage

Prefixes are used during autoconfiguring step to load Settings, while trying to avoid name clashes with user defined Settings. By default, *omniconf.prefix* will be loaded from the environment and cli arguments, by looking for `OMNICONF_PREFIX` and `--omniconf-prefix` respectively. In these settings, *omniconf* is the prefix.

To change the used during autoconfiguring, do the following:

```
from omniconf import omniconf_load
omniconf_load(config_registry=configs, autoconfigure_prefix="application")
```

The above example will set the prefix to *application*, which will cause autoconfiguring to look for `APPLICATION_PREFIX` and `--application-prefix` instead. Good if you don't want to leak that you're using omniconf to your users.

6.4 Backend prefix usage

Backends may allow a prefix to be defined. By default, this setting is loaded from the `omniconf.prefix` key (see previous section). If defined, this value is passed to all available backends, and will influence how they will load Config values.

For instance. if `omniconf.prefix` is not set, *`EnvBackend`* will load `some.setting` from the `SOME_SETTING` environment variable. If `omniconf.prefix` is set to `app`, the value is loaded from `APP_SOME_SETTING` instead. See the *Supported backends* section for which Backends allow a prefix to be configured, and how this changes the loading of values.

6.5 Prefix usage examples

Working with prefixes can be a little tricky. The thing to keep in mind is that there are two prefix types, one that is used during the autoconfigure step where the backends are initialized (the autoconfiguration prefix), and one that is used when loading the configuration (the backend prefix).

Given this code snippet:

```
from omniconf import omniconf_load, config, setting

setting("db.url", required=True)
omniconf_load(autoconfigure_prefix="test")

print config("db.url")
```

A step-by-step analysis:

1. The setting *db.url* is defined and marked as required.
2. Autoconfiguration is started and the *autoconfigure_prefix* is defined as 'test'.
    a. During autoconfiguration, by default *omniconf.prefix* will be looked up. Because we override *autoconfigure_prefix*, *test.prefix* is looked up instead.
    b. The contents of *test.prefix* is used by certain backends (*`EnvBackend`* in this example) to determine where they should look for their settings.
3. Config values are loaded, and the backend prefix is used to determine how it should be loaded.

6.5.1 Example 1

```
$ python test.py

Traceback (most recent call last):
...
omniconf.exceptions.UnconfiguredSettingError: No value was configured for db.url
```

An error is raised because we don't set any config values at all, and *db.url* is marked as required.

6.5.2 Example 2

```
$ TEST_DB_URL=bla python test.py
Traceback (most recent call last):
...
omniconf.exceptions.UnconfiguredSettingError: No value was configured for db.url
```

An error is raised because we set *TEST_DB_URL*, but no backend prefix has been configured. The value of *db.url* is looked up in *DB_URL* which is not set.

6.5.3 Example 3

```
$ TEST_PREFIX=OTHER OTHER_DB_URL=foo python test.py
foo
```

The backend prefix is set to *OTHER*. This means that the setting for *db.url* is looked up in *OTHER_DB_URL*, which is also set.

6.5.4 Example 4

```
$ DB_URL=foo python test.py
foo
```

No backend prefix is set. This means that the setting for *db.url* is looked up in *DB_URL*, which is also set.

6.6 Outputting usage information

To output `argparse`-like usage information based on `Setting` objects contained in a `SettingRegistry`, use the `show_usage()` function.

`omniconf.`**`show_usage`**(*setting_registry=None*, *name=None*, *top_message=None*, *bottom_message=None*, *out=None*, *exit=0*)

Prints usage information based on `Setting` objects in the given `SettingRegistry`. If no *setting_registry* is specified, the default `SettingRegistry` is used.

If no *name* is specified, *sys.argv[0]* is used. Additionally, a header and footer message may be supplied using *top_message* and *bottom_message* message respectively.

By default the usage information is output to *sys.stderr*. This can be overidden by specifying a different File-like object to *out*.

By default, this function will call *sys.exit* and stop the program with exit code 0. This can be overridden by a specifying different value to *exit*. Set to False to not exit.

For instance, the output for this piece of code:

```
from omniconf import setting, show_usage

setting("verbose", _type=bool, default=False, help="Enable verbose mode.")
setting("section1.setting", help="An optional setting")
setting("section1.other_setting", help="A different optional setting.")
setting("section2.setting", required=True, help="A required setting.")

show_usage(name="usage_example")
```

Looks like this:

```
usage: usage_example [--verbose] [--section1-other_setting SOS]
                     [--section1-setting SS] --section2-setting SS

optional arguments:
--verbose              Enable verbose mode.

section1:
--section1-other_setting SOS
                       A different optional setting.
--section1-setting SS
                       An optional setting

section2:
--section2-setting SS
                       A required setting.
```

An user who wants to show usage information, usually specifies a command line flag like `--help`. To detect this, omniconf provides a convenience method:

omniconf.**help_requested**()
: Returns True if *-h* or *–help* was specified on the command line.

Two other methods are also provided, one to detect a version flag, and one to detect any flag:

omniconf.**version_requested**()
: Returns True if *-v* or *–version* was specified on the command line.

omniconf.**flag_requested**(*flags*)
: Returns True if the specified list of flags were specified on the command line.

CHAPTER 7

Indices and tables

- genindex
- search

Index

H

J

O

S

U

V

Y