CONFspirator

Adrian Turjak

Sep 05, 2020

CONTENTS:

An offshoot of OpenStack's oslo.config with a focus on nested configuration groups, and the ability to use yaml and toml instead of flat ini files.

CONFspirator doesn't include any command-line integrations currently so you will need to add a command to your application to export a generated config using the built in functions.

It does have support for loading in config files, or a preloaded config dictionary against your config group tree.

The library's focus is on in-code defaults and config field validation, while giving you a lot of power when dealing with nesting, dynamic config loading for plugins, and useful overlay logic.

It allows you to define sane defaults, document your config, validate the values when loading it in, and provides useful ways of working with that config during testing.

CHAPTER

ONE

GETTING STARTED WITH CONFSPIRATOR

1.1 Installation

```
pip install confspirator
```

1.2 Usage

First lets put together a simple ConfigGroup, and register some config values:

```
# ./my_app/config/root.py
from confspirator import groups, fields

root_group = groups.ConfigGroup(
    "my_app", description="The root config group.")
root_group.register_child_config(
    fields.StrConfig(
        "top_level_config",
        help_text="Some top level config on the root group.",
        default="some_default",
    )
)
```

Then maybe let's make a second group, but in another file to keep things clear:

```
# ./my_app/config/sub_section.py
from confspirator import groups, fields
from my_app.config import root

sub_group = groups.ConfigGroup(
    "sub_section", description="A sub group under the root group.")
sub_group.register_child_config(
    fields.BoolConfig(
        "bool_value",
        help_text="some boolean flag value",
        default=True,
    )
)

root.root_group.register_child_config(sub_group)
```

Now we want to load in our config against this group definition and check the values:

```
# ./my_app/config/__init__.py
import confspirator
from my_app.config import root

CONF = confspirator.load_file(
    root.root_group, "/etc/my_app/conf.yaml")
```

Assuming your config file looks like:

```
# String - Some top level config on the root group.
top_level_config: some_default
# A sub group under the root group.
sub_section:
  # Boolean - some boolean flag value
  bool_value: true
```

Then in your application code you can pull those values out and use them:

```
# ./my_app/do_thing.py
from my_app.config import CONF

print(CONF.top_level_config)
print(CONF.sub_section.bool_value)
```

CHAPTER

TWO

CONFIGURATION FIELDS IN CONFSPIRATOR

CONFspirator supports multiple different fields for configuration, ranging from simple strings and ints, all the way to dictionaries and lists, with some special cases like ports, hostnames, and uri's.

2.1 Options for all fields

These options are supported for all config fields in addition to the specific ones the field might have.

2.1.1 Options

name

The config field's name. It is always the first parameter when defining a config field.

help_text

An explanation of how the option is used or what it is for.

default

The default value of the option.

required

If a value must be supplied for this option and cannot be empty or blank.

test_default

A default for when running in test mode.

sample_default

A default for sample config files.

unsafe_default (unused currently)

If the default value must be overridden.

Once implemented will warn or raise an error when a certain flag is given if the default value hasn't been overridden.

secret (unused currently)

If the value should be obfuscated in log output.

Once implemented will be used to help provide information about what values to obfuscated in logs.

deprecated_location

Deprecated dot separated location. Acts like an alias to where the config used to be. e.g. 'service.group1.old_name'

deprecated_for_removal

Indicates whether this opt is planned for removal in a future release.

deprecated_reason

Indicates why this opt is planned for removal in a future release. Silently ignored if deprecated_for_removal is False.

deprecated_since

indicates which release this opt was deprecated in. Accepts any string, though valid version strings are encouraged. Silently ignored if deprecated_for_removal is False

advanced (unused currently)

A bool True/False value if this option has advanced usage and is not normally used by the majority of users.

2.2 StrConfig

Simple string based config.

2.2.1 Options

name

The config's name.

choices (optional)

Optional sequence of either valid values or tuples of valid values with descriptions.

quotes (optional)

If True and string is enclosed with single or double quotes, will strip those quotes.

regex (optional)

Optional regular expression (string or compiled regex) that the value must match on an unanchored search.

ignore_case (optional)

If True case differences (uppercase vs. lowercase) between 'choices' or 'regex' will be ignored.

max_length (optional)

If positive integer, the value must be less than or equal to this parameter.

2.2.2 Example usage

```
config_group.register_child_config(
    fields.StrConfig(
        "my_string_config",
        help_text="Some useful help text.",
        required=True,
        default="stuff",
    )
)
```

2.3 BoolConfig

Simple boolean based config.

2.3.1 Options

name

The config's name.

2.3.2 Example usage

```
config_group.register_child_config(
    fields.BoolConfig(
        "my_boolean_config",
        help_text="Some useful help text.",
        required=True,
        default=False,
    )
)
```

2.4 IntConfig

Simple int based config.

2.4.1 Options

name

The config's name.

min (optional)

Minimum value the integer can take.

max (optional)

Maximum value the integer can take.

2.4.2 Example usage

```
config_group.register_child_config(
    fields.IntConfig(
        "my_int_config",
        help_text="Some useful help text.",
        required=True,
        default=6,
        min=1,
        max=10,
    )
)
```

2.5 FloatConfig

Simple float based config.

2.5.1 Options

name

The config's name.

min (optional)

Minimum value the float can take.

max (optional)

Maximum value the float can take.

2.5.2 Example usage

```
config_group.register_child_config(
    fields.FloatConfig(
        "my_float_config",
        help_text="Some useful help text.",
        required=True,
        default=6.4,
        min=1.2,
        max=10.9,
    )
)
```

2.6 ListConfig

A list config, with a configurable type for items.

2.6.1 Options

name

The config's name.

item_type (optional)

Type of items in the list (see `confspirator.types`). If not set will default to a list of strings.

2.6.2 Example usage

```
config_group.register_child_config(
    fields.ListConfig(
        "my_list_config",
        help_text="Some useful help text.",
        required=True,
        default=["stuff", "things"],
    )
)
```

2.7 DictConfig

A dict config, with a configurable type for values.

2.7.1 Options

name

The config's name.

value_type (optional)

Type of values in the dict (see `confspirator.types`). If not set will default to strings.

check_value_type (optional)

If value is already dict, should we check value type.

is_json (optional)

If True and value is string, will parse as json.

2.7.2 Example usage

```
config_group.register_child_config(
    fields.DictConfig(
        "my_dict_config",
        help_text="Some useful help text.",
        required=True,
        default={"stuff": "things"},
    )
)
```

2.8 IPConfig

IP address config.

2.8.1 Options

name

The config's name.

version (optional)

One of either `4`, `6`, or `None` to specify either version.

2.8.2 Example usage

```
config_group.register_child_config(
    fields.IPConfig(
        "my_ip_config",
        help_text="Some useful help text.",
        required=True,
        default=0.0.0.0,
        version=4,
    )
)
```

2.9 PortConfig

Config for a TCP/IP port number. Ports can range from 0 to 65535.

2.9.1 Options

name

The config's name.

min (optional)

Minimum value the port can take.

max (optional)

Maximum value the port can take.

choices (optional)

Sequence of valid values.

2.9.2 Example usage

```
config_group.register_child_config(
    fields.PortConfig(
        "my_port_config",
        help_text="Some useful help text.",
        required=True,
        default=222,
        min=2000,
        max=9999,
    )
)
```

2.10 HostNameConfig

Config for a hostname. Only accepts valid hostnames.

2.10.1 Options

name

The config's name.

2.10.2 Example usage

```
config_group.register_child_config(
    fields.HostNameConfig(
        "my_hostname_config",
        help_text="Some useful help text.",
        required=True,
        default="prod.cluster.thing.net",
    )
)
```

2.11 HostAddressConfig

Option for either an IP or a hostname.

2.11.1 Options

name

The config's name.

version (optional)

One of either `4`, `6`, or `None` to specify either version.

2.11.2 Example usage

```
config_group.register_child_config(
    fields.HostAddressConfig(
        "my_hostaddress_config",
        help_text="Some useful help text.",
        required=True,
        default="prod.cluster.thing.net",
    )
)
```

2.12 URIConfig

Option for either a URI.

2.12.1 Options

name

The config's name.

max_length (optional)

If positive integer, the value must be less than or equal to this parameter.

schemes (optional)

List of valid URI schemes, e.g. ‘https’, ‘ftp’, ‘git’.

2.12.2 Example usage

```
config_group.register_child_config(
    fields.URIConfig(
        "my_url_config",
        help_text="Some useful help text.",
        required=True,
        default="https://example.com",
        schemes["https", "http"]
    )
)
```

CHAPTER

THREE

EXAMPLE CONFIG GENERATION

CONFspirator supports generated config files from the built config tree. These will populate all the fields, and if a default value is supplied will put that in place.

To use this functionality you must supply your root config group, and call the function with a file location. CONFspirator does not supply any CLI support for this, so you will want to build a command into your application or project to import your root config group, and call the generation function.

Assuming the config group as shown in the getting started page:

```
# ./my_app/config/root.py
from confspirator import groups, fields

root_group = groups.ConfigGroup(
    "my_app", description="The root config group.")
root_group.register_child_config(
    fields.StrConfig(
        "top_level_config",
        help_text="Some top level config on the root group.",
        default="some_default",
    )
)

# ./my_app/config/sub_section.py
from confspirator import groups, fields
from my_app.config import root

sub_group = groups.ConfigGroup(
    "sub_section", description="A sub group under the root group.")
sub_group.register_child_config(
    fields.BoolConfig(
        "bool_value",
        help_text="some boolean flag value",
        default=True,
    )
)

root.root_group.register_child_config(sub_group)
```

You would call the generation logic as follows:

```
# ./my_app/commands.py

import confspirator
from my_app.config import root
```

(continues on next page)

(continued from previous page)

```
def create_config():
    confspirator.create_example_config(
        root.root_group, "conf.yaml")
```

This would produce a yaml config example that looks like the following:

```
# String - Some top level config on the root group.
top_level_config: some_default
# A sub group under the root group.
sub_section:
  # Boolean - some boolean flag value
  bool_value: true
```

Alternatively if you wanted `toml` instead, you can simply change the file extension and the exporter will pick that up. Or if you want to use an extension other than `yaml` or `toml` you can explicitly set `output_format` to either `yaml` or `toml`, and the file extension will be ignored:

```
confspirator.create_example_config(
    root.root_group, "my_app.conf", output_format="toml")
```

In any case, if by extension or explicit output format `toml` is set, your generated example config will look as follows:

```
[my_app]
# String - Some top level config on the root group.
top_level_config = "some_default"

# A sub group under the root group.
[my_app.sub_section]
  # Boolean - some boolean flag value
  bool_value = true
```

For complicated nested configs yaml tends to be easier to deal with, but for people with a preference for `ini` style configs toml does provide a good option that still allows nesting.

CHAPTER

FOUR

CONFSPIRATOR IN YOUR UNIT TESTS

You often need ways to override config when running tests, so CONFspirator provides some powerful ways to do that even for the most complicated of nested configs.

4.1 Setting test defaults

The first smart thing to do is set the `test_default` value on a given config field for something you want to be a global test default value. This is useful for values you rarely need to change, and this means you can put your test defaults in the same places you define you config and put your actual defaults.

An example of this might be:

```
config_group.register_child_config(
    fields.StrConfig(
        "my_string_config",
        help_text="Some useful help text.",
        required=True,
        default="stuff",
        test_default="test_specific_stuff",
    )
)
```

To ensure CONFspirator uses the `test_default` you will need to put it into `test_mode` when running the load config functions:

```
CONF = confspirator.load_file(
    config_group, "/etc/my_app/conf.yaml", test_mode=True)
```

That does mean you will need some way to decide when loading config if your application is running in `test_mode`. This will vary depending on your framework, or unit testing tools, but as an example, in Django you could do it as follows:

```
test_mode = False
if "test" in sys.argv:
    test_mode = True

CONF = confspirator.load_file(
    config_group, "/etc/my_app/conf.yaml", test_mode=test_mode)
```

4.2 Overriding config for test cases

Often in unit or functional testing you need ways to override config for the duration of a test, a whole set of tests, or even in different phases of a test.

As such CONFspirator provides an all powerful `modify_conf` function that alows you to selectively alter your config entity for the needed scope.

Note: `modify_conf` assumes your test case classes inherit from `unittest.TestCase`. If they do not, then this will not work.

A simple example:

```
import confspirator

from my_app.config import CONF


@confspirator.modify_conf(
    CONF,
    {
        "my_app.top_level_config": [
            {"operation": "override", "value": "a new value"}
        ],
    }
)
class BasicTests(TestCase):

    def test_top_level_config(self):
        self.assertEqual(CONF.top_level_config, "a new value")
```

It can also be used to decorate a single test function:

```
import confspirator

from my_app.config import CONF


class BasicTests(TestCase):

    @confspirator.modify_conf(
        CONF,
        {
            "my_app.top_level_config": [
                {"operation": "override", "value": "a new value"}
            ],
        }
    )
    def test_top_level_config(self):
        self.assertEqual(CONF.top_level_config, "a new value")
```

Or even a section when using the `with` keyword:

```
import confspirator

from my_app.config import CONF
```

(continues on next page)

(continued from previous page)

```
class BasicTests(TestCase):
    def test_top_level_config(self):
        with confspirator.modify_conf(
            CONF,
            {
                "my_app.top_level_config": [
                    {"operation": "override", "value": "a new value"}
                ],
            },
        ):
            self.assertEqual(CONF.top_level_config, "a new value")
```

4.2.1 parameters for modify_conf

`modify_conf` takes two argument which can be used positionally, or as keywords.

conf

This should be the loaded config entity and will be either an instance of `GroupNamespace` or in advanced cases `LazyLoadedGroupNamespace`.

operations

This is a dictionary of config values as dot separated paths to a list of operations.

It is possible to alter multiple config values at the same time, and run multiple operations on each. Operations will run in the order supplied and can be chained together (e.g. add a value to the start and end of a list).

Here is what a more complex example may look like:

```
operations={
    "my_app.api_settings.item_list_option": [
        {"operation": "remove", "value": "option1"},
        {"operation": "append", "value": "option15"},
    ],
    "my_app.api_settings.boolean_flag_option": [
        {"operation": "override", "value": False},
    ],
}
```

Available operations per config type

- **value:**
  - override
- **list:**
  - override
  - preprend

  - append
  - remove
- **dict:**
  - override
  - update
  - delete
  - overlay
- **GroupNamespace:**
  - override
  - overlay

Overlay is essentially a dict merge, where any keys present in the overlaying dictionay will be inserted or will override the ones in the target.

CHAPTER

FIVE

ADVANCED USAGE

5.1 Dynamic Group Naming

Sometimes a group can't be named ahead of time and needs to be named dynamically based on some other input. A good example of this might be because the group is related to a given class which may be subclassed and its name needs to be unique at time of registration rather than unique at time of definition.

A DynamicNameConfigGroup looks almost like a normal group, but it needs to have the name set before being registered or it will raise an error:

```
# ./my_app/plugins/dynamic_plugin.py
from confspirator import groups, fields

class SomePluginClass(object):

    config_group = groups.DynamicNameConfigGroup(
        description="A dynamicly named sub group under the root group.",
        children=[
            fields.BoolConfig(
                "bool_value",
                help_text="some boolean flag value",
                default=True,
            ),
        ]
    )

# ./my_app/plugins/loading.py
from my_app.config import root

def setup_plugin_config(plugin_class):
    config_copy = plugin_class.config_group.copy()
    config_copy.set_name(plugin_class.__class__.__name__)
    root.root_group.plugins_group.register_child_config(config_copy)
```

In the above example we are also first making a copy of the group so that we don't set a name on the group associated with the plugin class. The new copy will be a deep copy, and will include any subgroups if there are any.

5.2 Extending Existing Groups

Sometimes you would like to reuse and extend an existing config group. In those cases the extend function will allow you to make a deep copy of another config group, and add additional children to it (either fields or further sub-groups).

The original reason for this feature was for extending class associated config and being able to inherit the parent config as a base starting point:

```
# ./my_app/plugins/dynamic_plugin.py
from confspirator import groups, fields

class SomeOtherPluginClass(SomePluginClass):

    config_group = SomePluginClass.config_group.extend(
        children=[
            fields.StrConfig(
                "my_string_config",
                help_text="Some useful help text.",
                required=True,
                default="stuff",
            ),
        ]
    )
```

This could also be used outside of class contexts when building generic config groups as a base for other elements of your app, and having specific groups extend the base one for the extra parts they need. It helps minimise duplication if used right.

Note: When extending you can also pass a list of strings to the *remove_children* function parameter to have the keys given removed from the children of the group.

5.3 Overlaying Groups

Loaded config groups have the ability to have another group or a dictionary overlaid on top of them, with the returned object being a deep copy of the target overlaid by the given group or dictionary.

This is essentially a depth first dictionary update, where only keys in the given group or dictionary will be overridden inside the target group.

It can be as simple as just merging two groups together:

```
merged_config = CONF.plugins.defaults.overlay(
    CONF.plugins.SpecificPlugin)
```

But this is most effective when the configs are designed to be overlaid cleanly for the purposes of overriding defaults, or some other scenario where a base config group needs some of its values overridden by config defined elsewhere.

The original use case for this feature was to allow defining a default for all instances of a given class, and having an easy way to overlay more specific config on top of the defaults for different places that class was used, to have that config passed down to the code that needed it in the most specific state:

```
def get_action_config(action_name, task_type):
    try:
        action_defaults = CONF.workflow.action_defaults.get(action_name)
```

(continues on next page)

(continued from previous page)

```
    except KeyError:
        return {}

    try:
        task_conf = CONF.workflow.tasks[task_type]
        return action_defaults.overlay(task_conf.actions[action_name])
    except KeyError:
        return action_defaults
```

5.4 Lazy Loading Groups

In rather weird cases where plugins or application startup may register more configuration that can't be parsed right away, you can make a group lazy load itself when first accessed.

This might be useful where one part of your config will define what plugins are enabled, or needed to start the application at all, but during that start up or loading process more configs will be registered.

Having a group be lazy loaded is as simple as setting a parameter when making the group:

```
lazy_loaded_group = groups.ConfigGroup("lazy_loaded", lazy_load=True)
```

Any fields or subgroups registered to this group before it is first access via the loaded config will be included in the loaded config. Up until first access the loaded group namespace retains a link back to the group definition, the loaded config file, and environment variables, which are processed on first access into a fully loaded group namespace.