glorpen-config Documentation

Release 2.2.0-56-gc027813

Arkadiusz Dzięgiel

Apr 27, 2022

Contents

Config framework for Your projects - with validation, interpolation and value normalization. It can even generate default config with help texts!

CHAPTER 1

Official repositories

GitHub: https://github.com/glorpen/glorpen-config

BitBucket: https://bitbucket.org/glorpen/glorpen-config

GitLab: https://gitlab.com/glorpen/glorpen-config

CHAPTER 2

Features

You can:

- define configuration schema inside Python app
- convert configuration values to Python objects
- validate user provided data
- use interpolation to fill config values
- generate example configuration with help text

2.1 How to load data

You can use `Reader` to read values from arbitrary source and then pass it to `glorpen.config.Config`:

```
from glorpen.config.translators.yaml import YamlReader
from glorpen.config import Config

config = Config(String())
config.get(YamlReader("example.yaml").read())
```

or with use of `glorpen.config.Translator`:

```
from glorpen.config.translators.yaml import YamlReader
from glorpen.config import Config, Translator

translator = Translator(Config(String()))
translator.read(YamlReader("example.yaml"))
```

`glorpen.config.Config.get()` accepts anything that is supported by underlying config schema so you can pass `dict` or custom objects.

2.2 Interpolation

You can reuse values from config with dotted notation, eg: `{{ path.to.value }}`.

```
project:
   path: "/tmp"
   cache_path: "{{ project.path }}/cache"
```

See field documentation to find where interpolation is supported.

2.3 Normalization and validation

Each field type has own normalization rules, eg. for `glorpen.config.fields.log.LogLevel`:

```
logging: DEBUG
```

`config.get(data)` would yield value `10` as in `logging.DEBUG`.

Additionally it will raise exception if invalid value is provided.

2.4 Optional and default values

Each field can have default value. If no value is given in config but default one is set, it will be used instead.

Default values should be already Python values, eg. `int`, `str`, objects.

CHAPTER 3

Contents

3.1 Example usage

3.1.1 Using fields

Your first step should be defining configuration schema:

```
import logging
import glorpen.config.fields.simple as f
from glorpen.config.fields.log import LogLevel

project_path = "/tmp/project"

spec = f.Dict({
  "project_path": f.Path(default=project_path),
  "project_cache_path": f.Path(default="{{ project_path }}/cache"),
  "logging": fl.LogLevel(default=logging.INFO),
  "database": f.String(),
  "sources": f.Dict({
      "some_param": f.String(),
      "some_path": f.Path(),
  }),
  "maybe_string": f.Variant([
      f.String(),
      f.Number()
  ])
})
```

Example yaml config:

```
logging: "DEBUG"
database: "mysql://...."
sources:
  some_param: "some param"
```

(continues on next page)

(continued from previous page)

```
  some_path: "/tmp"
maybe_string: 12
```

Then you can create `glorpen.config.Config` instance:

```
from glorpen.config import Config
import glorpen.config.loaders as loaders

loader = loaders.YamlLoader(filepath=config_path)
cfg = Config(loader=loader, spec=spec).finalize()

cfg.get("sources.some_param") #=> 'some param'
cfg.get("project_path") #=> '/tmp/project'
cfg.get("project_cache_path") #=> '/tmp/project/cache'
cfg.get("logging") #=> 10
cfg.get("maybe_string") #=> 12
```

3.1.2 Creating custom fields

Custom field class should extend *`glorpen.config.fields.base.Field`* or `glorpen.config.fields.base.FieldWithDefault`.

`glorpen.config.fields.base.Field.make_resolvable()` method should register normalizer functions which later will be called in registration order. Each value returned by normalizer is passed to next one. After chain end value is returned as config value.

Returned `glorpen.config.fields.base.ResolvableObject` instance is resolved before passing it to next normalizer.

If value passed to normalizator is invalid it should raise `glorpen.config.exceptions.ValidationError`. Sometimes value can be lazy loaded - it is represented as `glorpen.config.fields.base.ResolvableObject`. You can get real value by using `glorpen.config.fields.base.resolve()`.

```
class MyValue(object):
   def __init__(self, value):
      super(MyValue, self).__init__()
      self.value = value

class MyField(Field):

    def to_my_value(self, value, config):
        return MyValue(value)

    def is_value_supported(self, value):
        return True

    def make_resolvable(self, r):
        r.on_resolve(self.to_my_value)
```

The last thing is to use prepared custom field in configuration spec.

3.2 glorpen.config API Documentation

3.2.1 glorpen.config

glorpen.config.**__version__**
Current package version.

3.2.2 glorpen.config.config

class glorpen.config.config.**Config**
Config validator and normalizer.

3.2.3 glorpen.config.fields.base

class glorpen.config.fields.base.**Field**(*validators=None*)
Single field in configuration file.

Custom fields should implement own normalizer/interpolation by overriding corresponding methods.

To add custom validation based on whole config object use `validator()`.

get_dependencies(*normalized_value*)
Find parts that can be interpolated and return required deps. Should check only own data, no nested fields.

interpolate(*normalized_value*, *values*) → None
Should replace data in normalized_value with interpolated one. Called only when `get_dependencies` finds something.

3.2.4 glorpen.config.fields.simple

class glorpen.config.fields.simple.**Any**(*validators=None*)
Field that accepts any value.

class glorpen.config.fields.simple.**Dict**(*schema=None*, *keys=None*, *values=None*, *check_keys=False*, ***kwargs*)
Converts values to `collections.OrderedDict`

Supports setting whole schema (specific keys and specific values) or just keys type and values type.

Keys can be interpolated if `keys` param supports it.

__init__(*schema=None*, *keys=None*, *values=None*, *check_keys=False*, ***kwargs*)
To set specific schema pass dict to schema argument: `{"param1": SomeField()}`.

To specify keys and values type use keys and values arguments: `Dict(keys=String(), values=Number())`.

class glorpen.config.fields.simple.**List**(*schema*, *check_values=False*, ***kwargs*)
Converts value to list.

class glorpen.config.fields.simple.**Number**(*validators=None*)
Converts value to numbers.

class glorpen.config.fields.simple.**Path**(**args*, *split_by='.'*, *left_char='{'*, *right_char='}'*, ***kwargs*)
Converts given value to disk path.

class glorpen.config.fields.simple.**PathObj**(**args*, *split_by='.'*, *left_char='{'*, *right_char='}'*, ***kwargs*)

Converts value to `pathlib.Path` object.

class glorpen.config.fields.simple.**String**(**args*, *split_by='.'*, *left_char='{'*, *right_char='}'*, ***kwargs*)

Converts value to string.

class glorpen.config.fields.simple.**Variant**(*schema*, **args*, ***kwargs*)

Converts value to normalized state using one *Field* chosen from multiple provided.

To allow blank values you have to pass child field with enabled blank values. First field which supports value (`Field.is_value_supported()`) will be used to convert it.

3.2.5 glorpen.config.fields.log

3.2.6 glorpen.config.fields.version

3.2.7 glorpen.config.translators.base

3.2.8 glorpen.config.translators.yaml

3.2.9 glorpen.config.exceptions

exception glorpen.config.exceptions.**ConfigException**

Base exception for config errors.

exception glorpen.config.exceptions.**TraceableConfigException**(*exception*)

Exception for improved readability - uses `ValidationError` to provide full path to field with error.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

g

Index

Symbols

A

C

D

F

G

I

L

N

P

S

T

V