confmodel Documentation

Release 0.2.0-py2.7.egg

Praekelt Foundation

June 05, 2014

Contents

Release 0.2.0-py2.7.egg

Note: This documentation is not yet complete. It will be updated and improved as we work on confmodel.

Thank you for your attention.

confmodel is a tool for accessing, validating, and documenting configuration parameters. Config specifications are written as Python classes with specialised field attributes (similar to Django forms) and then instantiated with configuration data. The configuration is validated and fields are available as parameters on the config object (similar to Django models, although read-only).

CHAPTER 1

Installation

```
$ pip install confmodel
```

CHAPTER 2

Documentation

2.1 Using confmodel

2.1.1 Defining your configuration

A config specification is a subclass of `Config` with some field attributes.

```
from confmodel import Config
from confmodel.fields import ConfigInt, ConfigText

class MyConfig(Config):
    """
    This is a demo config specification.

    It's important to write a docstring for the class and to put helpful
    information into the mandatory ``doc`` parameter in each field.
    """

    incantation = ConfigText("The incantation to recite.", required=True)
    magic_number = ConfigInt("A magic number.", default=42)
```

As a bonus, confmodel generates ReST docstrings for your config classes, suitable both for runtime introspection and inclusion in Sphinx documentation.

```
>>> print MyConfig.__doc__
This is a demo config specification.

It's important to write a docstring for the class and to put helpful
information into the mandatory ``doc`` parameter in each field.

Configuration options:

:param str incantation:

    The incantation to recite.

:param int magic_number:

    A magic number.
```

Example rendered documentation:

class MyConfig

This is a demo config specification.

It's important to write a docstring for the class and to put helpful information into the mandatory `doc` parameter in each field.

Configuration options:

Parameters

- **incantation** (*str*) – The incantation to recite.
- **magic_number** (*int*) – A magic number.

2.1.2 Accessing config data

Once the specification has been defined, it can be used to access configuration data acquired from some arbitrary source. A config specification class can be instantiated with a `dict` [1] containing keys that match the field attributes.

```
>>> config = MyConfig({'incantation': 'Open sesame!'})
>>> config.incantation
'Open sesame!'
>>> config.magic_number
42
```

The data is validated when the config object is instantiated, so you'll know immediately if something is wrong.

```
>>> config = MyConfig({})  # No configuration data.
Traceback (most recent call last):
    ...
ConfigError: Missing required config field 'incantation'

>>> config = MyConfig({'incantation': 'Open sesame!', 'magic_number': 'six'})
Traceback (most recent call last):
    ...
ConfigError: Field 'magic_number' could not be converted to int.
```

2.1.3 Further information

Sometimes it's necessary for a config field to refer to other fields to find or construct its value, particularly as systems evolve over time. See *Field fallbacks* for ways to do this.

2.2 Advanced features

2.2.1 Field fallbacks

Sometimes it's necessary for a config field to refer to other fields to find or construct its value, particularly as systems evolve over time. This is managed in a flexible way using `FieldFallback` objects.

```
from confmodel import Config
from confmodel.fields import ConfigText
from confmodel.fallbacks import SingleFieldFallback
```

[1] More generally, any *IConfigData* provider can be used. A `dict` is just the simplest and most convenient for many cases.

```
class SimpleFallbackConfig(Config):
    """
    This config specification demonstrates the use of a SingleFieldFallback.
    """

    incantation = ConfigText(
        "The incantation to recite. (Falls back to the 'magic_word' field.)",
        required=True, fallbacks=[SingleFieldFallback("magic_word")])
    magic_word = ConfigText("*DEPRECATED* The magic word to utter.")
```

The above specification requires the `incantation` field, but of that's not present the `magic_word` field will be used instead. Validation will fail if neither is present.

```
>>> SimpleFallbackConfig({u'incantation': u'foo'}).incantation
u'foo'
>>> SimpleFallbackConfig({u'magic_word': u'please'}).incantation
u'please'
>>> SimpleFallbackConfig({}).incantation
Traceback (most recent call last):
    ...
ConfigError: Missing required config field 'incantation'
```

A field used as a fallback is still a normal field in every way.

```
>>> print SimpleFallbackConfig({u'incantation': u'foo'}).magic_word
None
>>> SimpleFallbackConfig({u'magic_word': u'please'}).magic_word
u'please'
```

Multiple fallbacks

Multiple fallbacks may be used for a single field by listing then in order of preference.

```
>>> from confmodel import Config
>>> from confmodel.fields import ConfigText
>>> from confmodel.fallbacks import SingleFieldFallback
>>> class MultiFallbackConfig(Config):
...     """
...     This config specification demonstrates the use of multiple fallbacks.
...     """
...     incantation = ConfigText(
...         "The incantation to recite."
...         " (Falls back to the 'magic_word' and 'galdr' fields.)",
...         required=True, fallbacks=[
...             SingleFieldFallback("magic_word"),
...             SingleFieldFallback("galdr"),
...         ])
...     magic_word = ConfigText("*DEPRECATED* The magic word to utter.")
...     galdr = ConfigText("*DEPRECATED* Runes to chant.")

>>> MultiFallbackConfig({u'incantation': u'foo'}).incantation
u'foo'
>>> MultiFallbackConfig({u'magic_word': u'please'}).incantation
u'please'
>>> MultiFallbackConfig({u'galdr': u'heyri jotnar'}).incantation
u'heyri jotnar'
>>> MultiFallbackConfig({
```

```
...     u'magic_word': u'please',
...     u'galdr': u'heyri jotnar',
... }).incantation
u'please'
>>> MultiFallbackConfig({}).incantation
Traceback (most recent call last):
    ...
ConfigError: Missing required config field 'incantation'
```

Fallbacks with defaults

Default values for fallbacks are ignored [2] and the field's default value is used as a last resort if no fallback values are found.

```
>>> from confmodel import Config
>>> from confmodel.fields import ConfigText
>>> from confmodel.fallbacks import SingleFieldFallback
>>> class FallbackDefaultsConfig(Config):
...     """
...     This config specification demonstrates fallbacks with defaults.
...     """
...     incantation = ConfigText(
...         "The incantation to recite. (Falls back to the 'magic_word' field.)",
...        default=u"xyzzy", fallbacks=[SingleFieldFallback("magic_word")])
...     magic_word = ConfigText(
...         "*DEPRECATED* The magic word to utter.", default=u"plugh")

>>> FallbackDefaultsConfig({u'incantation': u'foo'}).incantation
u'foo'
>>> FallbackDefaultsConfig({u'magic_word': u'please'}).incantation
u'please'
>>> FallbackDefaultsConfig({}).incantation
u'xyzzy'
```

Format string fallback

For more complex fallbacks, `FormatStringFieldFallback` can be used.

```
>>> from confmodel import Config
>>> from confmodel.fields import ConfigInt, ConfigText
>>> from confmodel.fallbacks import FormatStringFieldFallback
>>> class FormatFallbackConfig(Config):
...     """
...     This config specification demonstrates format string fallbacks.
...     """
...     url_base = ConfigText(
...         "A host:port pair.", required=True, fallbacks=[
...             FormatStringFieldFallback(u"{host}:{port}", ["host", "port"]),
...         ])
...     host = ConfigText("A hostname.")
...     port = ConfigInt("A network port.")

>>> FormatFallbackConfig({u'url_base': u'example.com:80'}).url_base
u'example.com:80'
```

[2] Although custom `FieldFallback` subclasses may override this behaviour.

```
>>> FormatFallbackConfig({u'host': u'example.org', u'port': 8080}).url_base
u'example.org:8080'
>>> FormatFallbackConfig({u'host': u'example.net'}).url_base
Traceback (most recent call last):
    ...
ConfigError: Missing required config field 'url_base'
```

2.2.2 Custom fallbacks

If your needs aren't met by the standard fallback classes, you can subclass `FieldFallback` to implement custom behaviour.

TODO: Write something about custom fallback classes.

2.2.3 Static fields

TODO: Write something about static fields.

2.3 API documentation

2.3.1 `confmodel` module

The top-level namespace contains some convenience imports.

Members

`__version__`
Version number.

class `Config`(*config_data*, *static=False*)
Config object.

Configuration options:

`post_validate`()
Subclasses may override this to provide cross-field validation.

Implementations should raise `ConfigError` if the configuration is invalid (by calling `raise_config_error()`, for example).

`raise_config_error`(*message*)
Raise a `ConfigError` with the given message.

2.3.2 `confmodel.errors` module

Exception classes used in confmodel.

Members

exception ConfigError
Bases: exceptions.Exception

Base exception class for config-related errors.

2.3.3 confmodel.interfaces module

Interfaces used in confmodel.

Members

interface IConfigData
Interface for a config data provider.

This provides read-only access to some configuration data provider. The simplest implementation is a vanilla dict.

__contains__ (*field_name*)
Check for the existence of a config field.

This is identical to has_key() but is often more convenient to use.

Parameters **field_name** (*str*) – The name of the field to look up.

Returns True if a value exists for the given field_name, False otherwise.

has_key (*field_name*)
Check for the existence of a config field.

Parameters **field_name** (*str*) – The name of the field to look up.

Returns True if a value exists for the given field_name, False otherwise.

get (*field_name*, *default*)
Get the value of a config field.

Parameters

- **field_name** (*str*) – The name of the field to look up.
- **default** – The value to return if the requested field is not found.

Returns The value for the given field_name, or default if the field has not been specified.

2.4 Config field reference

2.4.1 ConfigField base class

class ConfigField (*doc*, *required=False*, *default=None*, *static=False*, *fallbacks=()*)
The base class for all config fields.

A config field is a descriptor that reads a value from the source data, validates it, and transforms it into an appropriate Python object.

Parameters

- **doc** (*str*) – Description of this field to be included in generated documentation.

- **required** (*bool*) – Set to `True` if this field is required, `False` if it is optional. Unless otherwise specified, fields are not required.
- **default** – The default value for this field if no value is provided. This is unused if the field is required.
- **static** (*bool*) – Set to `True` if this is a static field. See *Static fields* for further information.
- **fallbacks** – A list of `FieldFallback` objects to try if the value isn't present in the source data. See *Field fallbacks* for further information.

Subclasses of `ConfigField` are expected to override `clean()` to convert values from the source data to the required form. `clean()` is called during validation and also on every attribute access, so it should not perform expensive computation. (If expensive computation is necessary for some reason, the result should be cached.)

There are two special attributes on this descriptor:

field_type = None
A class attribute that specifies the field type in generated documentation. It should be a string, or `None` to indicate that the field type should remain unspecified.

name
An instance attribute containing the name bound to this descriptor instance. It is set by metaclass magic when a `Config` subclass is defined.

clean(*value*)
Clean and process a value from the source data.

This should be overridden in subclasses to handle different kinds of fields.

Parameters **value** – A value from the source data.

Returns A value suitable for Python code to use. This implementation merely returns the value it was given.

find_value(*config*)
Find a value in the source data, fallbacks, or field default.

Parameters **config** – `Config` object containing config data.

Returns The first value it finds.

get_doc()
Build documentation for this field.

A reST `:param:` field is generated based on the `name`, `doc`, and `field_type` attributes.

Returns A string containing a documentation section for this field.

get_value(*config*)
Get the cleaned value for this config field.

This calls `find_value()` to get the raw value and then calls `clean()` to process it, unless the value is `None`.

This method may be overridden in subclasses if `None` needs to be handled differently.

Parameters **config** – `Config` object containing config data.

Returns A cleaned value suitable for Python code to use.

present(*config*, *check_fallbacks=True*)
Check if a value for this field is present in the config data.

Parameters

- **config** – `Config` object containing config data.
- **check_fallbacks** (*bool*) – If `False`, fallbacks will not be checked. (This is used internally to determine whether to use fallbacks when looking up data.)

Returns `True` if the value is present in the provided data, `False` otherwise.

raise_config_error(*message_suffix*)
Raise a `ConfigError` referencing this field.

The text "Field '<field name>' <message suffix>" is used as the exception message.

Parameters **message_suffix** (*str*) – A string to append to the exception message.

Returns Doesn't return, but raises a `ConfigError`.

validate(*config*)
Check that the value is present if required and valid if present.

If the field is required but no value is found, a `ConfigError` is raised. Further validation is performed by calling `clean()` and the value is assumed to be valid if no exceptions are raised.

Parameters **config** – `Config` object containing config data.

Returns `None`, but exceptions are raised for validation failures.

2.4.2 confmodel.fields module

All standard config field classes live here.

TODO: Write docstrings for remaining more fields.

2.5 Field fallback reference

2.5.1 FieldFallback base class

class FieldFallback

field_present(*config*, *field_name*)
Check if a value for the named field is present in the config data.

Parameters

- **config** – `Config` instance containing config data.
- **field_name** (*str*) – Name of the field to look up.

Returns `True` if the value is present in the provided data, `False` otherwise.

2.5.2 confmodel.fallbacks module

All standard field fallback classes live here.

TODO: Write docstrings for remaining more fallbacks. TODO: Document base class separately?

C