settei Documentation

Release 0.5.0

Spoqa, Inc

Jun 11, 2019

Contents

Configuration utility for common Python applications and services. FYI, “settei” () means settings in Japanese. :)

CHAPTER 1

Loading a configuration is easy

Suppose you use Flask with Settei.

```
from flask import Flask
from settei import Configuration, config_property

class WebConfiguration(Configuration):
    """Load Configuration::

       [web]
       debug = true

    """

    #: debug option
    debug = config_property('web.debug', bool, default=False)


conf = WebConfiguration.from_path(pathlib.Path('.') / 'dev.toml')
app = Flask(__name__)


if __name__ == '__main__':
     app.run(debug=conf.debug)
```

1.1 `settei` — App object holding configuration

1.1.1 settei.base — Basic app object

New in version 0.2.0.

exception settei.base.**ConfigError**
The base exception class for errors releated to *Configuration* and *config_property()*.

New in version 0.4.0.

exception settei.base.**ConfigKeyError**
An exception class rises when there's no a configuration key. A subtype of *ConfigError* and KeyError.

New in version 0.4.0.

exception settei.base.**ConfigTypeError**
An exception class rises when the configured value is not of a type the field expects.

New in version 0.4.0.

class settei.base.**Configuration**(*config: Mapping[str, object] = {}*, ***kwargs*)
Application instance with its settings e.g. database. It implements read-only Mapping protocol as well, so you can treat it as a dictionary of string keys.

Changed in version 0.4.0: Prior to 0.4.0, it had raised Python's built-in KeyError on missing keys, but since 0.4.0 it became to raise *ConfigKeyError*, a subtype of KeyError, instead.

classmethod from_file(*file*) → settei.base.Configuration
Load settings from the given file and instantiate an *Configuration* instance from that.

Parameters **file** – the file object that contains TOML settings

Returns an instantiated configuration

Return type *Configuration*

classmethod from_path(*path: pathlib.Path*) → Configuration
Load settings from the given path and instantiate an *Configuration* instance from that.

Parameters **path** (pathlib.Path) – the file path that contains TOML settings

Returns an instantiated configuration

Return type *Configuration*

exception settei.base.**ConfigValueError**
An execption class rises when the configured value is somewhat invalid.

New in version 0.4.0.

exception settei.base.**ConfigWarning**
Warning category which raised when a default configuration is used instead due to missing required configuration.

class settei.base.**config_object_property**(*key: str*, *cls*, *docstring: str = None*, *recurse: bool = False*, ***kwargs*)
Similar to *config_property* except it purposes to reprsent more complex objects than simple values. It can be utilized as dependency injector.

Suppose a field declared as:

```
from werkzeug.contrib.cache import BaseCache

class App(Configuration):
    cache = config_object_property('cache', BaseCache)
```

Also a configuration:

```
[cache]
class = "werkzeug.contrib.cache:RedisCache"
host = "a.nodes.redis-cluster.local"
port = 6379
db = 0
```

The above instantiates the following object:

```
from werkzeug.contrib.cache import RedisCache
RedisCache(host='a.nodes.redis-cluster.local', port=6380, db=0)
```

There's a special field named `*` which is for positional arguments as well:

```
[cache]
class = "werkzeug.contrib.cache:RedisCache"
"*" = [
    "a.nodes.redis-cluster.local",
    6379,
]
db = 0
```

The above configuration is equivalent to the following Python code:

```
from werkzeug.contrib.cache import RedisCache
RedisCache('a.nodes.redis-cluster.local', 6380, db=0)
```

By default it doesn't recursively evaluate. For example, the following configuration:

```
[field]
class = "a:ClassA"
[field.value]
class = "b:ClassB"
[field.value.value]
class = "c:ClassC"
```

is evaluated to:

```
from a import ClassA
ClassA(value={'class': 'b:ClassB', 'value': {'class': 'c:ClassC'}})
```

If `recurse=True` option is provided, it evaluates nested tables too:

```
from a import ClassA
from b import ClassB
from c import ClassC

ClassA(value=ClassB(value=ClassC()))
```

Parameters

- **key** (`str`) – the dotted string of key path. for example `abc.def` looks up `config['abc']['def']`
- **cls** (`type`) – the allowed type of the configuration
- **docstring** (`str`) – optional documentation about the configuration. it will be set to `__doc__` attribute

- **recurse** (bool) – whether to evaluate nested tables as well. False by default
- **default** – keyword only argument. optional default value used for missing case. cannot be used with default_func at a time
- **default_func** (collections.abc.Callable) – keyword only argument. optional callable which returns a default value for missing case. it has to take an App mapping, and return a default value. cannot be used with default at a time
- **default_warning** (bool) – keyword only argument. whether to warn when default value is used. does not warn by default. this option is only available when default value is provided

New in version 0.4.0.

New in version 0.5.0: The recurse option.

class settei.base.**config_property**(*key: str*, *cls*, *docstring: str = None*, ***kwargs*)

Declare configuration key with type hints, default value, and docstring.

Parameters

- **key** (str) – the dotted string of key path. for example abc.def looks up config['abc']['def']
- **cls** (type) – the allowed type of the configuration
- **docstring** (str) – optional documentation about the configuration. it will be set to __doc__ attribute
- **default** – keyword only argument. optional default value used for missing case. cannot be used with default_func at a time
- **default_func** (collections.abc.Callable) – keyword only argument. optional callable which returns a default value for missing case. it has to take an App mapping, and return a default value. cannot be used with default at a time
- **default_warning** (bool) – keyword only argument. whether to warn when default value is used. does not warn by default. this option is only available when default value is provided

Changed in version 0.4.0: Prior to 0.4.0, it had raised Python's built-in KeyError on missing keys, but since 0.4.0 it became to raise *ConfigKeyError*, a subtype of KeyError, instead.

In the same manner, while prior to 0.4.0, it had raised Python's built-in TypeError when a configured value is not of a type it expects, but since 0.4.0 it became to raise *ConfigTypeError* instead. *ConfigTypeError* is also a subtype of TypeError.

docstring

(str) The propertly indented __doc__ string.

1.1.2 settei.presets — Richer presets for several frameworks

New in version 0.2.0.

settei.presets.celery — Preset for Celery

class settei.presets.celery.**WorkerConfiguration**(*config: Mapping[str, object] = {}*, ***kwargs*)

The application object mixin which holds configuration for Celery.

on_worker_loaded(*app*)
Be invoked when a Celery app is ready.

Parameters **app** (celery.Celery) – a ready celery app

worker_broker_url
The url of the broker used by Celery. See also Celery's and Kombu's docs about broker urls:

http://docs.celeryproject.org/en/latest/configuration.html#broker-url http://kombu.readthedocs.org/en/latest/userguide/connections.html#connection-urls

worker_config
(typing.Mapping[str, object]) The configuration maping for worker that will go to Celery.conf.

worker_result_backend
The backend used by Celery to store task results. See also Celery's docs about result backends:

http://docs.celeryproject.org/en/latest/configuration.html#celery-result-backend

worker_schedule
(typing.Mapping[str, typing.Mapping[str, object]]) The schedule table for Celery Beat, scheduler for periodic tasks.

There's some preprocessing before reading configuration. Since TOML doesn't have custom types, you can't represent timedelta or crontab values from the configuration file. To workaround the problem, it evaluates strings like 'f()' pattern if they are appeared in a schedule field.

For example, if the following configuration is present:

```
[worker.celerybeat_schedule.add-every-30-seconds]
task = "tasks.add"
schedule = "timedelta(seconds=30)"  # string to be evaluated
args = [16, 16]
```

it becomes translated to:

```
CELERYBEAT_SCHEDULE = {
    'add-every-30-seconds': {
        'task': 'tasks.add',
        'schedule': datetime.timedelta(seconds=30),  # evaluated!
        'args': (16, 16),
    },
}
```

Note that although timedelta and crontab is already present in the context, you need to import things if other types. It can also parse and evaluate the patterns like 'module.path:func()'.

Also args fields are translated from array to tuple.

See also Celery's docs about periodic tasks:

http://docs.celeryproject.org/en/latest/userguide/periodic-tasks.html

New in version 0.2.2.

settei.presets.flask — Preset for Flask apps

New in version 0.2.0.

class settei.presets.flask.**WebConfiguration**(*config: Mapping[str, object] = {}*, ***kwargs*)

on_web_loaded(*app: Callable*)
Be invoked when a WSGI app is ready.

Parameters **app** (flask.Flask, typing.Callable) – a ready wsgi/flask app

web_config
(typing.Mapping) The configuration maping for web that will go to flask.Flask.config.

web_debug
Whether to enable debug mode. On debug mode the server will reload itself on code changes, and provide a helpful debugger when things go wrong.

settei.presets.logging — Preset for logging configuration

New in version 0.2.0.

Preset for apps holding logging configuration. Logging can be configured through TOML file e.g.:

```
[logging]
version = 1

[logging.loggers.flask]
handlers = ["stderr"]

[logging.loggers."urllib.request"]
handlers = ["stderr"]

[logging.loggers.werkzeug]
handlers = ["stderr"]

[logging.handlers.stderr]
class = "logging.StreamHandler"
level = "INFO"
stream = "ext://sys.stderr"
```

class settei.presets.logging.**LoggingConfiguration**(*config: Mapping[str, object] = {}*, ***kwargs*)

Hold configuration for logging.

configure_logging() → None
Configure logging.

1.1.3 settei.version — Version data

New in version 0.2.0.

settei.version.**VERSION = '0.5.0'**
(str) The version string e.g. '1.2.3'.

settei.version.**VERSION_INFO = (0, 5, 0)**
(typing.Tuple[int, int, int]) The triple of version numbers e.g. (1, 2, 3).

1.2 Changlog

1.2.1 Version 0.5.0

Released on July 24, 2017.

- Added `recurse` option to *`config_object_property`*. If it's `True` nested tables are also evaluated. `False` by default for backward compatibility.

1.2.2 Verison 0.4.0

Released on May 14, 2017.

- *`config_object_property`* was added. It's a kind of dependency injection, but very limited version.
- *`ConfigError`*, *`ConfigKeyError`*, *`ConfigTypeError`*, and *`ConfigValueError`*.

  Prior to 0.4.0, *`Configuration`* had raised Python's built-in `KeyError` on missing keys, but since 0.4.0 it became to raise *`ConfigKeyError`*, a subtype of `KeyError`, instead.

  In the same manner, while prior to 0.4.0, it had raised Python's built-in `TypeError` when a configured value is not of a type it expects, but since 0.4.0 it became to raise *`ConfigTypeError`* instead. *`ConfigTypeError`* is also a subtype of `TypeError`.

1.2.3 Version 0.3.0

Released on January 22, 2017.

- As tsukkomi is now abandoned, it's replaced by typeguard.

1.2.4 Version 0.2.2

Released on November 18, 2016. Note that the version 0.2.1 has never been released due to our mistake on versioning.

- *`WorkerConfiguration`* became to have *`worker_schedule`* config property to configure Celery beat — Celery's periodic tasks.

1.2.5 Version 0.2.0

Released on July 13, 2016.

- *`settei`* became a package (had been a module), which contains *`settei.base`* module.
- `settei.Configuration`, `settei.ConfigWarning`, and `settei.config_property` were moved to *`settei.base`* module. Although aliases for these previous import paths will be there for a while, we recommend to import them from *`settei.base`* mdoule since they are deprecated.
- Presets were introduced: *`settei.presets`*.
  - *`settei.presets.celery`* is for configuring Celery apps.
  - *`settei.presets.flask`* is for configuring Flask web apps.
  - *`settei.presets.logging`* is for configuring Python standard `logging` system.
- *`settei.version`* module was added.

- typeannotations was replaced by tsukkomi.
- Settei now requires pytoml 0.1.10 or higher. (It had required 0.1.7 or higher.)

1.2.6 Version 0.1.1

Released on April 15, 2016.

- *settei.base.config_property* became to support `typing.Union` type.

1.2.7 Version 0.1.0

Released on April 1, 2016. Initial release.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index