b3j0f.conf Documentation

Release 0.3.22

b3j0f

October 10, 2016

Table of Contents

1 ChangeLog 1

2 Indices and tables 7

3 Description 9

4 Links 11

5 Installation 13

6 Features 15

7 Example 19

8 Perspectives 23

9 Donation 25

CHAPTER 1

ChangeLog

1.1 0.3.21 (2016/10/05)

- fix default serialized value.

1.2 0.3.21 (2016/10/05)

- add an error message when using the Array object with a wrong item type.
- fix default svalue/value for param.

1.3 0.3.20 (2016/09/03)

- improve support of specific parser.

1.4 0.3.19 (2016/09/02)

- fix support of specific parser.

1.5 0.3.18 (2016/06/06)

- fix support of python3.4 with old style class.

1.6 0.3.17 (2016/06/05)

- fix support of python3.4.

1.7 0.3.16 (2016/06/05)

- fix reloading of modules.
- add the parameter modules in all configurable methods in order to load modules before use it.
- add the parameter rel for forcing configurable module reloading. Default is False.

1.8 0.3.15 (2016/06/02)

- fix compatibility with python>3.4.

1.9 0.3.14 (2016/04/10)

- Add support for default parameters automatically set in the configuration.

1.10 0.3.13 (2016/04/10)

- Add item type in the Array class.

1.11 0.3.12 (2016/04/10)

- save Configurable modules such as a list of modules and not a list of string.

1.12 0.3.11 (2016/04/05)

- add user/absolute path in file driver.
- avoid to set parameters when already given in constructor parameters.

1.13 0.3.10 (2016/04/05)

- fix UTs and subconfiguration.

1.14 0.3.9 (2016/04/03)

- improve the API in authorizing the use of Configuration, Category or Parameter when a configuration is requested.
- fix a bug when calling the applyconfiguration function with inheritance requirements between the default conf and a new conf.

1.15 0.3.8 (2016/04/02)

- add BOOL and ARRAY parameter type converters.
- fix parameter conversion from ptype.

1.16 0.3.7 (2016/04/02)

- add support of inheritance in updating model elements and get params from configuration.
- remove cleaned parameter in the methods ModelElement.copy and ModelElement.update.

1.17 0.3.6 (2016/04/02)

- fix bug while updating parameter ptype (new None values did change old consistent values).

1.18 0.3.5 (2016/04/02)

- fix bug while intercepting a configured object instanciation without resource reading.

1.19 0.3.4 (2016/03/30)

- simplify installation in using the package_data parameter in the setup.
- move etc to b3j0f/conf/data.

1.20 0.3.3 (2016/03/30)

- fix easy_install installation.

1.21 0.3.2 (2016/03/29)

- fix installation of the configuration files.

1.22 0.3.1 (2016/03/16)

- add support for recursive configuration of sub objects.
- simplify code.
- add the attribute keepstate which ensure sub objects are not reinstantiate if they already exist.

1.23 0.3.0 (2016/03/12)

- a Configurable inherits from an b3j0f.annotation.Annotation
- a configurable can inject configuration in function parameters.
- support xml files.
- add logger in Configurable.
- simplify the Logger configurable.
- support sub configuration.

1.24 0.2.5 (2016/02/20)

- fix installation via easy-install in adding the etc folder in the project.

1.25 0.2.4 (2016/01/11)

- add confpath parameter in order to import configurable configuration from a file.
- add ui package.

1.26 0.2.3 (2015/12/20)

- add support for python2.6.

1.27 0.2.2 (2015/12/16)

- add the function model.parser.serialize in order to easily serialiaze Param values.
- simplify driver API in order to make easier the development of new drivers.
- move the logging part from the Configurable class to the specific module configurable.logger.
- set inheritance to Configurable from b3j0f.annotation.PrivateCallInterceptor.
- remove decorator module.
- add foreigns attributes in Configurable which allows to add not specified parameters given by conf resources.
- add autoconf attribute in Configurable, getconfigurables and applyconfiguration functions.
- rename get_conf, set_conf, to_configure and apply_configuration to getconf, setconf, targets and applyconfiguration.
- add Configurable.safe attribute in order to execute configuration in an unsafe context if necessary.
- add the configurable Logger useful to ease management of complex logging needs.

1.28 0.2.1 (2015/10/29)

- add the module model.parser which contains all parser functions provided previously in the class Parameter.
- add serialized value in parameter.
- add the parser eval which evaluates a simple and safe python lambda body expression (without I/O functions).

1.29 0.2.0 (2015/10/28)

- simplify the global architecture in removing both module registry and ParamList.
- separate the module model to three dedicated modules: model.configuration, model.parameter, model.category.
- add model UTs.
- add parameter conf and type in Parameter in order to respectively set initialization parameter value with additional configuration data and force parameter type.
- add regex in parameter name.
- allow to configure parameter values which are configurables.
- add the property Parameter.error which equals an Exception if change of value fired an exception.
- add the module version in order to manage from one access point the project version number.

1.30 0.1.9 (2015/09/28)

- use b3j0f.utils.property.addproperties in order to reduce code lines.
- use the english date time format in the changelog file.

1.31 0.1.8 (2015/09/22)

- add reference to Configurable, ConfigurableRegistry, ConfDriver, Configuration, Category and Parameter in the main package.

1.32 0.1.7 (2015/07/22)

- fix bug about targets parameter.
- update README in fixing the example.

1.33 0.1.6 (2015/06/13)

- use the docs directory related to readthedocs requirements.

1.34 0.1.5 (2015/06/13)

- use shields.io badges in the README.

1.35 0.1.4 (2015/06/02)

- use B3J0F_CONF_DIR environment variable in order to get default FileConfDriver default path for given conf files. Otherwise, use '~/etc' path.

1.36 0.1.2 (2015/05/20)

- remove retrocompatibility with python2.6

1.37 0.1.1 (2015/05/20)

- add __all__ in modules and packages
- add base classes in packages
- fix UTs in all python versions but 2.6

1.38 0.1.0 (2015/05/20)

- commit first version with poor comments and documentation.
- watcher module does not work.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

CHAPTER 3

Description

Python object configuration library in reflective and distributed concerns.

CHAPTER 4

Links

- Homepage
- PyPI
- Documentation

CHAPTER 5

Installation

pip install b3j0f.conf

CHAPTER 6

Features

This library provides a set of object configuration tools in order to ease development of systems based on configuration resources such as files, DB documents, etc. in a reflexive, functional and distributed context.

Configuration processing is done is in 5 steps with only two mandatories:

1. optional : configuration model specification. You define which parameters you want to setup (default value, type, etc.).
2. optional : configuration driver selection. You define the nature of configuration resources (xml, json, ini, etc.).
3. mandatory : configuration resource edition. You define your configuration resources (files, DB documents, etc.).
4. optional : bind a configurable to python object(s). You bind a configurable to objects to setup for a reflexive use.
5. mandatory : apply configuration to a python object. You setup your python objects.

In order to improve reflexive concerns, this library has been developed with very strong flexible features available at runtime.

It is possible to custom every parts directly at runtime with high speed execution concerns.

You can customize:

- parsing functions.
- drivers.
- configuration process.
- configuration meta-model.

6.1 Configuration

The package b3j0f.conf.model provides Configuration class in order to inject configuration resources in a configurable class.

Configuration resources respect two levels of configuration same as the ini configuration model. Both levels are respectively:

- Category: permits to define a set of parameters unique by name.
- Parameter: couple of property name and value.

And all categories/parameters are embedded in a Configuration class.

Configuration and Categories are ordered dictionaries where key values are inner element names. Therefore, Parameter names are unique per Categories, and Category/Parameter names are unique per Configuration.

6.2 Parameter Overriding

In a dynamic and flexible way, one Configurable class can be bound to several Categories and Parameters. The choice is predetermined by the class itself. That means that one Configurable class can use several configuration resources and all/some/none Categories/Parameters in those resources. In such a way, the order is important because it permits to keep only last parameter values when parameters have the same name.

This overriding is respected by default with Configurable class inheritance. That means a sub class which adds a resource after its parent class resources indicates than all parameters in the final resources will override parameters in the parent class resources where names are the sames.

6.2.1 Name such as a regular expression

Even if a parameter uses a name, it is possible to use this name such as a regular expression name.

In this case, while reading a configuration from a resource, all configuration parameters which match the regular expression will use the same attributes as defined in the parameter with the regular expression. And a name which is the name from the configuration resource.

6.2.2 Parser

The package b3j0f.conf.parser provides parsers used to deserialize parameter values.

All parsers use special entries such as:

- %(lang:)expr%: string value of a programming language expression. `lang` is the target language (default is python) and `expr` is the expression to evaluate. For example, `%"testy"[:-1]%` equals `test`.
- @((rpath/)cname.)[history]pname: reference to another parameter value (thanks to the idea from the pypi config project) where:
  - `rpath`: optional resource path.
  - `cname`: optional category name.
  - `history`: optional parameter history in the arborescence from the (current) category. If not given, last parameter value is given.
  - `pname`: parameter name.

  For example: `@test` designates the final value of the parameter `test`. `@myconf.json/test` is the parameter `test` from the resource `myconf.json`. `@mycat...test` is the parameter `test` defined two categories before the category `mycat`.

If the parameter configuration value is of the format `=(lang:)expr`, the result is an evaluation of `expr` in the given language `lang`. For example, `=2` equals the integer 2. `=sys.maxsize` is the maxsize value from the sys module (thanks to the best effort property).

Finally, here are reserved keywords in an evaluated expression:

- `conf`: current configuration.
- `configurable`: current configurable.

Note: By default, `besteffort` and `safe` properties are enabled. They respectively permit to resolve absolute object path and avoid to call I/O functions. They can be changed in the resperive Configurable, like the expression execution scope value which contains by default both configuration and `configurable` noted above.

6.3 Configurable

A Configurable class is provided in the b3j0f.conf.configurable.core module. It permits to get parameters from configuration resources and setup them to a python object or in a method/function parameters.

It inherits from the `b3j0f.annotation.Annotation` class (annotation). That means it can be used such as an annotation (decorator with annotation functions). For example, bound Configurables to an object `o` are retrieved like this: `Configurable.get_annotations(o)`.

In order to perform a configuration setup, it uses:

- conf: configuration model.
- drivers: configuration resource drivers.
- scope: parsing execution scope.
- besteffort: boolean flag which aims to resolve automatically absolute object path. True by default.
- safe: boolean flag which aims to cancel calls to I/O functions. True by default.
- logger: logger which will handle all warning/error messages. None by default.

Annotated elements are bound and can all be (re)configured once in using the method applyconfiguration.

6.4 Driver

Drivers are the mean to parse configuration resources, such as files, DB documents, etc. from a configuration model provided by a Configurable object.

By default, conf drivers are able to parse json/ini/xml files. Those last use a relative path given by the environment variable `B3J0F_CONF_DIR` or from directories (in this order) `/etc`, `/usr/local/etc`, `~/etc`, `~/.config`, `~/config`, current execution directory or absolute path.

CHAPTER 7

Example

7.1 Bind configuration files to an object

Bind the configuration file `~/etc/myobject.conf` and `~/.config/myobject.conf` to a business class `MyObject` (the relative path `~/etc` can be changed thanks to the environment variable `B3J0F_CONF_DIR`).

The configuration file contains a category named `MYOBJECT` containing the parameters:

- `myattr` equals `'myvalue'`.
- `six` equals `6`.
- `twelve` equals `six * 2.0`.

Let the following configuration file `~/etc/myobject.conf` in ini format:

```
[MYOBJECT]
myattr = myvalue
twelve = =@six * 2.0
```

Let the following configuration file `~/.config/myobject.conf` in json format:

```
{
  "MYOBJECT": {
    "six": 6
  }
}
```

The following code permits to load upper configuration to a python object.

```
from b3j0f.conf import Configurable

# instantiate a business class
@Configurable(paths='myobject.conf')
class MyObject(object):
    pass

myobject = MyObject()

# assert attributes
assert myobject.myattr == 'myvalue'
assert myobject.six == 6
assert isinstance(myobject.six, int)
assert myobject.twelve == 12
assert isinstance(myobject.twelve, float)
```

The following code permits to load upper configuration to a python function.

```
from b3j0f.conf import Parameter, Array

# instantiate a business class and ensure twelve is converted into an integer and a string with comma
@Configurable(
  paths='myobject.conf',
  conf=[
    Parameter('default', value=len('default')),  # default value for the parameter default
    Parameter('twelve', ptype=int),
    Parameter('integers', svalue='1,2,3', ptype=Array(int))
)
def myfunc(myattr=None, six=None, twelve=None, default=None):  # Only None values will be setted by
    return myattr, six, twelve, default

myattr, six, twelve, default = myfunc(twelve=46)

# assert attributes
assert myobject.default == 7
assert myobject.myattr == 'myvalue'
assert myobject.six == 6
assert myobject.twelve == 46
assert isinstance(myobject.twelve, int)
assert myobject.integers == [1, 2, 3]
```

7.2 Class configuration

```
from b3j0f.conf import category
from b3j0f.conf import Configurable

# class configuration
@Configurable(conf=category('land', Parameter('country', value='fr')))
class World(object):
    pass

world = World()
assert world.country == 'fr'
```

7.3 One configuration with several objects

```
land = Configurable.get_annotations(world)[0]  # class method for retrieving Configurables

@land  # bind land to World2
class World2(object):
    pass

world2 = World2()

assert World2().country == 'fr'

class World3(object):
    pass

world3 = World3()
```

```
land.applyconfiguration([world3])  # apply land on world3 without annotated it

assert world3.country == 'fr'

world3s = (land(World3()) for _ in range(5))  # annotate a set of worlds

land.conf['land']['country'] = 'en'  # change default country code

land.applyconfiguration()  # update all annotated objects

for _world3 in world3s:
    assert _world3.country == 'en'

assert world2.country == 'en'

assert world3.country == 'fr'  # check not annotated element has not changed
```

7.4 Configure object constructor

```
land.autoconf = False  # disable autoconf

@land
class World4(object):
    def __init__(self, country=None):
        self.safecountry = country

world4 = World4()

assert not hasattr(world4, 'country')  # disabled auto conf side effect
assert world4.safecountry = 'en'

land.applyconfiguration()  # configure all land targets

assert world4.country == 'en'
```

7.5 Configure function parameters

```
@land
def getcountry(country=None):
    print(country)
    return country

assert getcountry() == 'en'

land['land']['country'] = 'fr'

assert getcountry() == 'fr'
```

7.6 Configure embedded objects

```
from b3j0f.conf import configuration, category, Parameter

class SubTest(object):
    pass

@Configurable(
    conf=(
        configuration(
            category('', Parameter('subtest', value=SubTest)),
# the prefix ':' refers (recursively) to a sub configuration for the parameter named with the suffix
            category(':subtest', Parameter('subsubtest', value=SubTest)),
            category(':subtset:subsubtest', Parameter('test', value=True))
        )
    )
)
class Test(object):
    pass

test = Test()

assert isinstance(test.subtest, SubTest)
assert isinstance(test.subtest.subsubtest)
assert not hasattr(test.subtest, 'test')
assert test.subtest.subsubtest.test is True

# and you can still apply configuration
Configurable.get_annotations(test)[0].conf[':subtset:subsubtest']['test'] = False
applyconfiguration(targets=[test])

assert test.subtest.subsubtest.test is False
```

7.7 Configure metaclasses

```
from six import add_metaclass

@Configurable(conf=Parameter('test', value=True))
class MetaTest(type):
    pass

@add_metaclass(MetaTest)
class Test(object):
    pass

test = Test()

assert test.test is True
```

CHAPTER 8

Perspectives

- wait feedbacks during 6 months before passing it to a stable version.
- Cython implementation.
- add GUI in order to ease management of multi resources and categorized parameters.

CHAPTER 9

Donation