zope.i18nmessageid Documentation

Release 4.0

Zope Foundation contributors

Jun 16, 2017

Contents

Contents:

CHAPTER 1

Using `zope.i18nmessageid`

Rationale

To translate any text, we must be able to discover the source domain of the text. A source domain is an identifier that identifies a project that produces program source strings. Source strings occur as literals in python programs, text in templates, and some text in XML data. The project implies a source language and an application context.

Messages and Domains

We can think of a source domain as a collection of messages and associated translation strings. The domain helps to disambiguate messages based on context: for instance, the message whose source string is `draw` means one thing in a first-person shooter game, and quite another in a graphics package: in the first case, the domain for the message might be `ok_corral`, while in the second it might be `gimp`.

We often need to create unicode strings that will be displayed by separate views. The view cannot translate the string without knowing its source domain. A string or unicode literal carries no domain information, therefore we use instances of the `Message` class. Messages are unicode strings which carry a translation source domain and possibly a default translation.

Message Factories

Messages are created by a message factory belonging to a given translation domain. Each message factory is created by instantiating a `MessageFactory`, passing the domain corresponding to the project which manages the corrresponding translations.

```
>>> from zope.i18nmessageid import MessageFactory
>>> factory = MessageFactory('myproject')
>>> foo = factory('foo')
>>> foo.domain
'myproject'
```

The Zope project uses the `zope` domain for its messages. This package exports an already-created factory for that domain:

```
>>> from zope.i18nmessageid import ZopeMessageFactory as _z_
>>> foo = _z_('foo')
>>> foo.domain
'zope'
```

Example Usage

In this example, we create a message factory and assign it to _. By convention, we use _ as the name of our factory to be compatible with translatable string extraction tools such as xgettext. We then call _ with a string that needs to be translatable:

```
>>> from zope.i18nmessageid import MessageFactory, Message
>>> _ = MessageFactory("futurama")
>>> robot = _(u"robot-message", u"${name} is a robot.")
```

Messages at first seem like they are unicode strings:

```
>>> robot == u'robot-message'
True
>>> isinstance(robot, unicode)
True
```

The additional domain, default and mapping information is available through attributes:

```
>>> robot.default == u'${name} is a robot.'
True
>>> robot.mapping
>>> robot.domain
'futurama'
```

The message's attributes are considered part of the immutable message object. They cannot be changed once the message id is created:

```
>>> robot.domain = "planetexpress"
Traceback (most recent call last):
...
TypeError: readonly attribute

>>> robot.default = u"${name} is not a robot."
Traceback (most recent call last):
...
TypeError: readonly attribute

>>> robot.mapping = {u'name': u'Bender'}
Traceback (most recent call last):
...
TypeError: readonly attribute
```

If you need to change their information, you'll have to make a new message id object:

```
>>> new_robot = Message(robot, mapping={u'name': u'Bender'})
>>> new_robot == u'robot-message'
```

```
True
>>> new_robot.domain
'futurama'
>>> new_robot.default == u'${name} is a robot.'
True
>>> new_robot.mapping == {u'name': u'Bender'}
True
```

Last but not least, messages are reduceable for pickling:

```
>>> callable, args = new_robot.__reduce__()
>>> callable is Message
True
>>> args == (u'robot-message',
...          'futurama',
...          u'${name} is a robot.',
...          {u'name': u'Bender'})
True

>>> fembot = Message(u'fembot')
>>> callable, args = fembot.__reduce__()
>>> callable is Message
True
>>> args == (u'fembot', None, None, None)
True
```

Pickling and unpickling works, which means we can store message IDs in a database:

```
>>> from pickle import dumps, loads
>>> pystate = dumps(new_robot)
>>> pickle_bot = loads(pystate)
>>> (pickle_bot,
...  pickle_bot.domain,
...  pickle_bot.default,
...  pickle_bot.mapping) == (u'robot-message',
...                          'futurama',
...                          u'${name} is a robot.',
...                          {u'name': u'Bender'})
True
>>> pickle_bot.__reduce__()[0] is Message
True
```

CHAPTER 2

zope.i18nmessageid API Reference

zope.i18nmessageid.message

CHAPTER 3

Hacking on `zope.i18nmessageid`

Getting the Code

The main repository for `zope.i18nmessageid` is in the Zope Foundation Github repository:

https://github.com/zopefoundation/zope.i18nmessageid

You can get a read-only checkout from there:

```
$ git clone https://github.com/zopefoundation/zope.i18nmessageid.git
```

or fork it and get a writeable checkout of your fork:

```
$ git clone git@github.com/jrandom/zope.i18nmessageid.git
```

The project also mirrors the trunk from the Github repository as a Bazaar branch on Launchpad:

https://code.launchpad.net/zope.i18nmessageid

You can branch the trunk from there using Bazaar:

```
$ bzr branch lp:zope.i18nmessageid
```

Working in a `virtualenv`

Installing

If you use the `virtualenv` package to create lightweight Python development environments, you can run the tests using nothing more than the `python` binary in a virtualenv. First, create a scratch environment:

```
$ /path/to/virtualenv --no-site-packages /tmp/hack-zope.i18nmessageid
```

Next, get this package registered as a "development egg" in the environment:

```
$ /tmp/hack-zope.i18nmessageid/bin/python setup.py develop
```

Running the tests

Run the tests using the build-in `setuptools` testrunner:

```
$ /tmp/hack-zope.i18nmessageid/bin/python setup.py test -q
running test
....................
----------------------------------------------------------------------
Ran 20 tests in 0.001s

OK
```

The `dev` command alias downloads and installs extra tools, like the `nose` testrunner and the `coverage` coverage analyzer:

```
$ /tmp/hack-zope.i18nmessageid/bin/python setup.py dev
$ /tmp/hack-zope.i18nmessageid/bin/nosetests
running nosetests
....................
----------------------------------------------------------------------
Ran 20 tests in 0.030s

OK
```

If you have the `coverage` pacakge installed in the virtualenv, you can see how well the tests cover the code:

```
$ /tmp/hack-zope.i18nmessageid/bin/nosetests --with coverage
running nosetests
....................
Name                          Stmts   Miss  Cover   Missing
-----------------------------------------------------------
zope.i18nmessageid                3      0   100%
zope.i18nmessageid.message       36      0   100%
-----------------------------------------------------------
TOTAL                            39      0   100%
----------------------------------------------------------------------
Ran 21 tests in 0.036s

OK
```

Building the documentation

`zope.i18nmessageid` uses the nifty `Sphinx` documentation system for building its docs. Using the same virtualenv you set up to run the tests, you can build the docs:

The `docs` command alias downloads and installs Sphinx and its dependencies:

```
$ /tmp/hack-zope.i18nmessageid/bin/python setup.py docs
...
$ bin/sphinx-build -b html -d docs/_build/doctrees docs docs/_build/html
...
build succeeded.
```

You can also test the code snippets in the documentation:

```
$ bin/sphinx-build -b doctest -d docs/_build/doctrees docs docs/_build/doctest
...
running tests...

Document: index
---------------
1 items passed all tests:
  17 tests in default
17 tests in 1 items.
17 passed and 0 failed.
Test passed.

Doctest summary
===============
   17 tests
    0 failures in tests
    0 failures in setup code
build succeeded.
```

Using `zc.buildout`

Setting up the buildout

`zope.i18nmessageid` ships with its own `buildout.cfg` file and `bootstrap.py` for setting up a development buildout:

```
$ /path/to/python2.6 bootstrap.py
...
Generated script '.../bin/buildout'
$ bin/buildout
Develop: '/home/jrandom/projects/Zope/BTK/i18nmessageid/.'
...
Generated script '.../bin/sphinx-quickstart'.
Generated script '.../bin/sphinx-build'.
```

Running the tests

You can now run the tests:

```
$ bin/test --all
Running zope.testing.testrunner.layer.UnitTests tests:
  Set up zope.testing.testrunner.layer.UnitTests in 0.000 seconds.
  Ran 702 tests with 0 failures and 0 errors in 0.000 seconds.
Tearing down left over layers:
  Tear down zope.testing.testrunner.layer.UnitTests in 0.000 seconds.
```

Using tox

Running Tests on Multiple Python Versions

tox is a Python-based test automation tool designed to run tests against multiple Python versions. It creates a virtualenv for each configured version, installs the current package and configured dependencies into each virtualenv, and then runs the configured commands.

zope.i18nmessageid configures the following tox environments via its tox.ini file:

- The py26, py27, py33, py34, and pypy environments builds a virtualenv with pypy, installs zope.i18nmessageid and dependencies, and runs the tests via python setup.py test -q.
- The coverage environment builds a virtualenv with python2.6, installs zope.i18nmessageid and dependencies, installs nose and coverage, and runs nosetests with statement coverage.
- The docs environment builds a virtualenv with python2.6, installs zope.i18nmessageid and dependencies, installs Sphinx and dependencies, and then builds the docs and exercises the doctest snippets.

This example requires that you have a working python2.6 on your path, as well as installing tox:

```
$ tox -e py26
GLOB sdist-make: .../zope.i18nmessageid/setup.py
py26 sdist-reinst: .../zope.i18nmessageid/.tox/dist/zope.i18nmessageid-4.0.2dev.zip
py26 runtests: commands[0]
...
----------------------------------------------------------------------
Ran 1341 tests in 0.477s

OK
_____________________________________ summary ______________________________________
py26: commands succeeded
congratulations :)
```

Running tox with no arguments runs all the configured environments, including building the docs and testing their snippets:

```
$ tox
GLOB sdist-make: .../zope.i18nmessageid/setup.py
py26 sdist-reinst: .../zope.i18nmessageid/.tox/dist/zope.i18nmessageid-4.0.2dev.zip
py26 runtests: commands[0]
...
Doctest summary
===============
678 tests
   0 failures in tests
   0 failures in setup code
   0 failures in cleanup code
build succeeded.
_____________________________________ summary ______________________________________
py26: commands succeeded
py27: commands succeeded
py32: commands succeeded
pypy: commands succeeded
coverage: commands succeeded
docs: commands succeeded
congratulations :)
```

Contributing to `zope.i18nmessageid`

Submitting a Bug Report

`zope.i18nmessageid` tracks its bugs on Github:

https://github.com/zopefoundation/zope.i18nmessageid/issues

Please submit bug reports and feature requests there.

Sharing Your Changes

Note: Please ensure that all tests are passing before you submit your code. If possible, your submission should include new tests for new features or bug fixes, although it is possible that you may have tested your new code by updating existing tests.

If have made a change you would like to share, the best route is to fork the Githb repository, check out your fork, make your changes on a branch in your fork, and push it. You can then submit a pull request from your branch:

https://github.com/zopefoundation/zope.i18nmessageid/pulls

If you branched the code from Launchpad using Bazaar, you have another option: you can "push" your branch to Launchpad:

```
$ bzr push lp:~jrandom/zope.i18nmessageid/cool_feature
```

After pushing your branch, you can link it to a bug report on Launchpad, or request that the maintainers merge your branch using the Launchpad "merge request" feature.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search