zope.dublincore Documentation

Release 4.1.2

Zope Foundation and Contributors

Jul 25, 2017

Contents

Contents:

CHAPTER 1

Using `zope.dublincore`

Dublin Core Properties

A dublin core property allows us to use properties from dublin core by simply defining a property as DCProperty.

```
>>> from zope.dublincore import property

>>> from zope.interface import implementer
>>> from zope.annotation.interfaces import IAttributeAnnotatable
>>> @implementer(IAttributeAnnotatable)
... class DC(object):
...     title   = property.DCProperty('title')
...     author  = property.DCProperty('creators')
...     authors = property.DCListProperty('creators')
>>> obj = DC()
>>> obj.title = u'My title'
>>> print(obj.title)
My title
```

Let's see if the title is really stored in dublin core:

```
>>> from zope.dublincore.interfaces import IZopeDublinCore
>>> print(IZopeDublinCore(obj).title)
My title
```

Even if a dublin core property is a list property we can set and get the property as scalar type:

```
>>> obj.author = u'me'
>>> print(obj.author)
me
```

DCListProperty acts on the list:

```
>>> obj.authors == (u'me',)
True
```

```
>>> obj.authors = [u'I', u'others']
>>> obj.authors == (u'I', u'others')
True
>>> print(obj.author)
I
```

Dublin Core metadata as content data

Sometimes we want to include data in content objects which mirrors one or more Dublin Core fields. In these cases, we want the Dublin Core structures to use the data in the content object rather than keeping a separate value in the annotations typically used. What fields we want to do this with can vary, however, and we may not want the Dublin Core APIs to constrain our choices of field names for our content objects.

To deal with this, we can use speciallized adapter implementations tailored to specific content objects. To make this a bit easier, there is a factory for such adapters.

Let's take a look at the simplest case of this to start with. We have some content object with a *title* attribute that should mirror the Dublin Core *title* field:

```
>>> @implementer(IAttributeAnnotatable)
... class Content(object):
...     title = u""
...     description = u""
```

To avoid having a discrepency between the *title* attribute of our content object and the equivalent Dublin Core field, we can provide a specific adapter for our object:

```
>>> from zope.dublincore import annotatableadapter

>>> factory = annotatableadapter.partialAnnotatableAdapterFactory(
...     ["title"])
```

This creates an adapter factory that maps the Dublin Core *title* field to the *title* attribute on instances of our *Content* class. Multiple mappings may be specified by naming the additional fields in the sequence passed to *partialAnnotatableAdapterFactory()*. (We'll see later how to use different attribute names for Dublin Core fields.)

Let's see what happens when we use the adapter.

When using the adapter to retrieve a field set to use the content object, the value stored on the content object is used:

```
>>> content = Content()
>>> adapter = factory(content)

>>> print(adapter.title)


>>> content.title = u'New Title'
>>> print(adapter.title)
New Title
```

If we set the relevant Dublin Core field using the adapter, the content object is updated:

```
>>> adapter.title = u'Adapted Title'
>>> print(content.title)
Adapted Title
```

Dublin Core fields which are not specifically mapped to the content object do not affect the content object:

```
>>> adapter.description = u"Some long description."
>>> print(content.description)

>>> print(adapter.description)
Some long description.
```

Using arbitrary field names

We've seen the simple approach, allowing a Dublin Core field to be stored on the content object using an attribute of the same name as the DC field. However, we may want to use a different name for some reason. The *partialAnnotatableAdapterFactory()* supports this as well.

If we call *partialAnnotatableAdapterFactory()* with a mapping instead of a sequence, the mapping is used to map Dublin Core field names to attribute names on the content object.

Let's look at an example where we want the *abstract* attribute on the content object to be used for the *description* Dublin Core field:

```
>>> @implementer(IAttributeAnnotatable)
... class Content(object):
...     abstract = u""
```

We can create the adapter factory by passing a mapping to *partialAnnotatableAdapterFactory()*:

```
>>> factory = annotatableadapter.partialAnnotatableAdapterFactory(
...     {"description": "abstract"})
```

We can check the effects of the adapter as before:

```
>>> content = Content()
>>> adapter = factory(content)

>>> print(adapter.description)


>>> content.abstract = u"What it's about."
>>> print(adapter.description)
What it's about.

>>> adapter.description = u'Change of plans.'
>>> print(content.abstract)
Change of plans.
```

Limitations

The current implementation has a number of limitations to be aware of; hopefully these can be removed in the future.

- Only simple string properties, like *title*, are supported. This is largely because other field types have not been given sufficient thought. Attempting to use this for other fields will cause a *ValueError* to be raised by *partialAnnotatableAdapterFactory()*.
- The CMF-like APIs are not supported in the generated adapters. It is not clear that these APIs are used, but content object implementations should be aware of this limitation.

Time annotators

Time annotators store the creation resp. last modification time of an object. We will use a simple `Content` class as our example.

```
>>> class Content(object):
...     created = None
...     modified = None
```

The annotations are stored on the `IZopeDublinCore` adapter. This dummy adapter reads and writes from/to the context object.

```
>>> from zope.component import provideAdapter
>>> from zope.dublincore.interfaces import IZopeDublinCore
>>> class DummyDublinCore(object):
...     def __init__(self, context):
...         self.__dict__['context'] = context
...
...     def __getattr__(self, name):
...         return getattr(self.context, name)
...
...     def __setattr__(self, name, value):
...         setattr(self.context, name, value)

>>> provideAdapter(DummyDublinCore, (Content,), IZopeDublinCore)
```

Created annotator

The created annotator sets creation and modification time to current time.

```
>>> content = Content()
```

It is registered for the `ObjectCreatedEvent`:

```
>>> from zope.dublincore import timeannotators
>>> timeannotators._NOW = 'NOW'
>>> from zope.component import provideHandler
>>> from zope.dublincore.timeannotators import CreatedAnnotator
>>> from zope.lifecycleevent.interfaces import IObjectCreatedEvent
>>> provideHandler(CreatedAnnotator, (IObjectCreatedEvent,))

>>> from zope.event import notify
>>> from zope.lifecycleevent import ObjectCreatedEvent
>>> notify(ObjectCreatedEvent(content))
```

Both `created` and `modified` get set:

```
>>> content.created
'NOW'
>>> content.modified
'NOW'
```

The created annotator can also be registered for (object, event):

```
>>> from zope.component import subscribers
>>> provideHandler(CreatedAnnotator, (None, IObjectCreatedEvent,))
```

```
>>> content = Content()
>>> ignored = subscribers((content, ObjectCreatedEvent(content)), None)
```

Both `created` and `modified` get set this way, too:

```
>>> content.created
'NOW'
>>> content.modified
'NOW'
```

Modified annotator

The modified annotator only sets the modification time to current time.

```
>>> content = Content()
```

It is registered for the `ObjectModifiedEvent`:

```
>>> from zope.dublincore.timeannotators import ModifiedAnnotator
>>> from zope.lifecycleevent.interfaces import IObjectModifiedEvent
>>> provideHandler(ModifiedAnnotator, (IObjectModifiedEvent,))

>>> from zope.lifecycleevent import ObjectModifiedEvent
>>> notify(ObjectModifiedEvent(content))
```

Only `modified` gets set:

```
>>> print(content.created)
None
>>> content.modified
'NOW'
```

The modified annotator can also be registered for (object, event):

```
>>> provideHandler(ModifiedAnnotator, (None, IObjectModifiedEvent,))
>>> content = Content()
>>> ignored = subscribers((content, ObjectModifiedEvent(content)), None)
```

`modified` gets set, this way, too:

```
>>> print(content.created)
None
>>> content.modified
'NOW'
```

CHAPTER 2

`zope.dublincore` API

CHAPTER 3

Hacking on `zope.dublincore`

Getting the Code

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

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

You can get a read-only checkout from there:

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

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

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

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

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

You can branch the trunk from there using Bazaar:

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

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.dublincore
```

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

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

Running the tests

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

```
$ /tmp/hack-zope.dublincore/bin/python setup.py test
running test
........
----------------------------------------------------------------------
Ran 80 tests in 0.000s

OK
```

If you have the `nose` package installed in the virtualenv, you can use its testrunner too:

```
$ /tmp/hack-zope.dublincore/bin/easy_install nose
...
$ /tmp/hack-zope.dublincore/bin/nosetests
.......
----------------------------------------------------------------------
Ran 80 tests in 0.000s

OK
```

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

```
$ /tmp/hack-zope.dublincore/bin/easy_install nose coverage
...
$ /tmp/hack-zope.dublincore/bin/nosetests --with coverage
running nosetests
.....................................................................................
Name                                        Stmts   Miss  Cover   Missing
-------------------------------------------------------------------------
zope/dublincore.py                              0      0   100%
zope/dublincore/annotatableadapter.py          73     73   100%
zope/dublincore/browser.py                      0      0   100%
zope/dublincore/browser/metadataedit.py        21     21   100%
zope/dublincore/creatorannotator.py            24     24   100%
zope/dublincore/dcsv.py                        92     92   100%
zope/dublincore/dcterms.py                     65     65   100%
zope/dublincore/interfaces.py                  72     72   100%
zope/dublincore/property.py                    66     66   100%
zope/dublincore/timeannotators.py              27     27   100%
zope/dublincore/xmlmetadata.py                173    173   100%
zope/dublincore/zopedublincore.py             198    !98   100%
-------------------------------------------------------------------------
TOTAL                                         811    811   100%
----------------------------------------------------------------------
Ran 86 tests in 0.000s

OK
```

Building the documentation

`zope.dublincore` 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:

```
$ /tmp/hack-zope.dublincore/bin/easy_install Sphinx
...
$ 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
...

Doctest summary
===============
   73 tests
    0 failures in tests
    0 failures in setup code
build succeeded.
Testing of doctests in the sources finished, look at the  \
    results in _build/doctest/output.txt.
```

Using `zc.buildout`

Setting up the buildout

`zope.dublincore` 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/dublincore/.'
...
Generated script '.../bin/sphinx-quickstart'.
Generated script '.../bin/sphinx-build'.
```

Running the tests

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 400 tests with 0 failures and 0 errors in 0.366 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.dublincore 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.dublincore and dependencies, and runs the tests via python setup.py test -q.
- The coverage environment builds a virtualenv with python2.6, installs zope.dublincore, installs nose and coverage, and runs nosetests with statement coverage.
- The docs environment builds a virtualenv with python2.6, installs zope.dublincore, 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.interface/setup.py
py26 sdist-reinst: .../zope.interface/.tox/dist/zope.interface-4.0.2dev.zip
py26 runtests: commands[0]
..........
----------------------------------------------------------------------
Ran 400 tests in 0.152s

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.interface/setup.py
py26 sdist-reinst: .../zope.interface/.tox/dist/zope.interface-4.0.2dev.zip
py26 runtests: commands[0]
...
Doctest summary
===============
  73 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.dublincore

Submitting a Bug Report

zope.dublincore tracks its bugs on Github:

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