zope.annotation Documentation

Release 4.0.1

Zope Foundation contributors

Oct 16, 2018

Contents

CHAPTER 1

Object Annotations

1.1 Annotation factories

There is more to document about annotations, but we'll just sketch out a scenario on how to use the annotation factory for now. This is one of the easiest ways to use annotations – basically you can see them as persistent, writable adapters.

First, let's make a persistent object we can create annotations for:

```
>>> from zope.interface import Interface
>>> from zope.interface import implementer
>>> class IFoo(Interface):
...     pass
>>> from zope.annotation.interfaces import IAttributeAnnotatable
>>> @implementer(IFoo, IAttributeAnnotatable)
... class Foo(object):
...     pass
```

We directly say that `Foo` implements `interfacesIAttributeAnnotatable` here. In practice this is often done in ZCML, using the `implements` subdirective of the `content` or `class` directive.

Now let's create an annotation for this:

```
>>> from zope.component import adapts
>>> from zope.interface import Attribute
>>> class IBar(Interface):
...     a = Attribute('A')
...     b = Attribute('B')
>>> from zope import component
>>> @implementer(IBar)
... class Bar(object):
...     adapts(IFoo)
...     def __init__(self):
...         self.a = 1
...         self.b = 2
```

Note that the annotation implementation does not expect any arguments to its `__init__`. Otherwise it's basically an adapter.

Now, we'll register the annotation as an adapter. To do this we use the *factory()* function provided by `zope.annotation`:

```
>>> from zope.component import provideAdapter
>>> from zope.annotation import factory
>>> provideAdapter(factory(Bar))
>>> from zope.component import provideAdapter
>>> from zope.annotation.attribute import AttributeAnnotations
>>> provideAdapter(AttributeAnnotations)
```

Note that we do not need to specify what the adapter provides or what it adapts - we already do this on the annotation class itself.

Now let's make an instance of `Foo`, and make an annotation for it.

```
>>> foo = Foo()
>>> bar = IBar(foo)
>>> bar.a
1
>>> bar.b
2
```

We'll change `a` and get the annotation again. Our change is still there:

```
>>> bar.a = 3
>>> IBar(foo).a
3
```

Of course it's still different for another instance of `Foo`:

```
>>> foo2 = Foo()
>>> IBar(foo2).a
1
```

What if our annotation does not provide what it adapts with `adapts`? It will complain:

```
>>> class IQux(Interface):
...     pass
>>> @implementer(IQux)
... class Qux(object):
...     pass
>>> provideAdapter(factory(Qux))
Traceback (most recent call last):
...
TypeError: Missing 'zope.component.adapts' on annotation
```

It's possible to provide an annotation with an explicit key. (If the key is not supplied, the key is deduced from the annotation's dotted name, provided it is a class.)

```
>>> class IHoi(Interface):
...     pass
>>> @implementer(IHoi)
... class Hoi(object):
...     adapts(IFoo)
>>> provideAdapter(factory(Hoi, 'my.unique.key'))
```

(continues on next page)

(continued from previous page)

```
>>> isinstance(IHoi(foo), Hoi)
True
```

1.2 Location

Annotation factories are put into the location hierarchy with their parent pointing to the annotated object and the name to the dotted name of the annotation's class (or the name the adapter was registered under):

```
>>> foo3 = Foo()
>>> new_hoi = IHoi(foo3)
>>> new_hoi.__parent__
<Foo object at 0x...>
>>> new_hoi.__name__
'my.unique.key'
>>> import zope.location.interfaces
>>> zope.location.interfaces.ILocation.providedBy(new_hoi)
True
```

Please notice, that our Hoi object does not implement ILocation, so a location proxy will be used. This has to be re-established every time we retrieve the object

(Guard against former bug: proxy wasn't established when the annotation existed already.)

```
>>> old_hoi = IHoi(foo3)
>>> old_hoi.__parent__
<Foo object at 0x...>
>>> old_hoi.__name__
'my.unique.key'
>>> zope.location.interfaces.ILocation.providedBy(old_hoi)
True
```

1.3 LocationProxies

Suppose your annotation proxy provides ILocation.

```
>>> class IPolloi(Interface):
...     pass
>>> @implementer(IPolloi, zope.location.interfaces.ILocation)
... class Polloi(object):
...     adapts(IFoo)
...     __name__ = __parent__ = 0
>>> provideAdapter(factory(Polloi, 'my.other.key'))
```

Sometimes you're adapting an object wrapped in a LocationProxy.

```
>>> foo4 = Foo()
>>> import zope.location.location
>>> wrapped_foo4 = zope.location.location.LocationProxy(foo4, None, 'foo4')
>>> located_polloi = IPolloi(wrapped_foo4)
```

At first glance it looks as if located_polloi is located under wrapped_foo4.

```
>>> located_polloi.__parent__ is wrapped_foo4
True
>>> located_polloi.__name__
'my.other.key'
```

but that's because we received a LocationProxy

```
>>> type(located_polloi).__name__
'LocationProxy'
```

If we unwrap located_polloi and look at it directly, we'll see it stores a reference to the real Foo object

```
>>> from zope.proxy import removeAllProxies
>>> removeAllProxies(located_polloi).__parent__ == foo4
True
>>> removeAllProxies(located_polloi).__name__
'my.other.key'
```

CHAPTER 2

zope.annotation API

2.1 Framework Interfaces

These interfaces define the source and targets for adaptation under the zope.annotation framework:

Annotations store arbitrary application data under package-unique keys.

interface zope.annotation.interfaces.**IAnnotatable**

Marker interface for objects that support storing annotations.

This interface says "There exists an adapter to an IAnnotations for an object that implements *IAnnotatable*".

Classes should not directly declare that they implement this interface. Instead they should implement an interface derived from this one, which details how the annotations are to be stored, such as *IAttributeAnnotatable*.

interface zope.annotation.interfaces.**IAnnotations**

Extends: *zope.annotation.interfaces.IAnnotatable*

Stores arbitrary application data under package-unique keys.

By "package-unique keys", we mean keys that are are unique by virtue of including the dotted name of a package as a prefix. A package name is used to limit the authority for picking names for a package to the people using that package.

For example, when implementing annotations for storing Zope Dublin-Core meta-data, we use the key:

```
"zope.app.dublincore.ZopeDublinCore"
```

__nonzero__()

Test whether there are any annotations

Must be identical to __bool__()

__bool__()

Test whether there are any annotations

Must be identical to __nonzero__()

__getitem__(*key*)
Return the annotation stored under key.

Raises KeyError if key not found.

get(*key*, *default=None*)
Return the annotation stored under key, or default if not found.

__setitem__(*key*, *value*)
Store annotation under key.

In order to avoid key collisions, users of this interface must use their dotted package name as part of the key name.

__delitem__(*key*)
Removes the annotation stored under key.

Raises a KeyError if the key is not found.

__iter__()
Return an iterator for the keys in the container.

__contains__(*key*)
Return True if 'key' is in the container, else False.

items()
Return '(key, value)' pairs for the keys in the container.

interface zope.annotation.interfaces.**IAttributeAnnotatable**
Extends: *zope.annotation.interfaces.IAnnotatable*

Marker indicating that annotations can be stored on an attribute.

This is a marker interface giving permission for an *IAnnotations* adapter to store data in an attribute named *__annotations__*.

2.2 Attribute-Based Annotations

The default adapter implementation uses a special attribute, `__annotations__`, on the annotated object:

class zope.annotation.attribute.**AttributeAnnotations**(*obj*, *context=None*)
Bases: _abcoll.MutableMapping

Store annotations on an object

Store annotations in the *__annotations__* attribute on a *IAttributeAnnotatable* object.

get(*key*, *default=None*)
See zope.annotation.interfaces.IAnnotations

keys() → list of D's keys

Because setting an attribute is somewhat intrusive (as opposed to storing annotations elsewhere), this adapter requires that its context implment *zope.annotation.interfaces.IAttributeAnnotatable* to signal that this attribute can be used.

2.3 Factories

Annotation factory helper

zope.annotation.factory.**factory**(*factory*, *key=None*)
Adapter factory to help create annotations easily.

CHAPTER 3

Hacking on `zope.annotation`

3.1 Getting the Code

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

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

You can get a read-only checkout from there:

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

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

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

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

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

You can branch the trunk from there using Bazaar:

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

3.2 Working in a `virtualenv`

3.2.1 Installing

If you use the `virtualenv` package to create lightweight Python development environments, you can work with your checkout using a virtualenv. First, create the virtualenv:

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

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

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

3.2.2 Running the tests

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

```
$ /tmp/hack-zope.annotation/bin/python setup.py -q test -q
running test
............
----------------------------------------------------------------------
Ran 11 tests in 0.000s

OK
```

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

```
$ /tmp/hack-zope.annotation/bin/python setup.py dev
$ /tmp/hack-zope.annotation/bin/nosetests
running nosetests
..........
----------------------------------------------------------------------
Ran 12 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.annotation/bin/nosetests --with coverage
............
Name                          Stmts   Miss  Cover   Missing
-----------------------------------------------------------
zope.annotation                   4      0   100%
zope.annotation.attribute        59      0   100%
zope.annotation.factory          28      0   100%
zope.annotation.interfaces       15      0   100%
-----------------------------------------------------------
TOTAL                           106     35    67%
----------------------------------------------------------------------
Ran 12 tests in 2.166s

OK
```

3.2.3 Building the documentation

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

(continues on next page)

(continued from previous page)

```
...
build succeeded.
```

You can also test the code snippets in the documentation:

```
$ /tmp/hack-zope.annotation/bin/sphinx-build -b doctest -d docs/_build/doctrees docs␣
↪docs/_build/doctest
...
running tests...

Document: narrative
------------------
1 items passed all tests:
  54 tests in default
54 tests in 1 items.
54 passed and 0 failed.
Test passed.

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

3.3 Using `zc.buildout`

3.3.1 Setting up the buildout

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

```
$ /path/to/python2.7 bootstrap.py
...
Generated script '.../bin/buildout'
$ bin/buildout
Develop: '/path/to/annotation/.'
...
Generated script '.../bin/sphinx-quickstart'.
Generated script '.../bin/sphinx-build'.
```

3.3.2 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 11 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.
```

3.3.3 Building the documentation

The `zope.annotation` buildout installs the Sphinx scripts required to build the documentation, including testing its code snippets:

```
$ cd docs
$ PATH=../bin:$PATH make doctest html
sphinx-build -b doctest -d _build/doctrees . _build/doctest
running tests...

Document: narrative
------------------
1 items passed all tests:
  54 tests in default
54 tests in 1 items.
54 passed and 0 failed.
Test passed.

Doctest summary
===============
   54 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.
sphinx-build -b html -d _build/doctrees   . _build/html
...
build succeeded.

Build finished. The HTML pages are in docs/_build/html.
```

3.4 Using `tox`

3.4.1 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.annotation` 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.annotation` and dependencies, and runs the tests via `python setup.py test -q`.
- The `nobtree` environment builds a `virtualenv` with Python 2.7, installs `zope.annotation` and its minimal dependencies (no `persistent` or `BTrees`), and runs the tests via `python setup.py test -q`.
- The `coverage` environment builds a `virtualenv` with `python2.7`, installs `zope.annotation` and dependencies, installs `nose` and `coverage`, and runs `nosetests` with statement coverage.
- The `docs` environment builds a virtualenv with `python2.7`, installs `zope.annotation` 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.annotation/setup.py
py26 sdist-reinst: .../zope.annotation/.tox/dist/zope.annotation-4.x.ydev.zip
py26 runtests: commands[0]
...
----------------------------------------------------------------------
Ran 11 tests in 0.000s

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.annotation/setup.py
py26 sdist-reinst: .../zope.annotation/.tox/dist/zope.annotation-4.0.2dev.zip
py26 runtests: commands[0]
...
Doctest summary
===============
  54 tests
   0 failures in tests
   0 failures in setup code
   0 failures in cleanup code
build succeeded.
_____________________________________ summary ______________________________________
py26: commands succeeded
py27: commands succeeded
py33: commands succeeded
py34: commands succeeded
pypy: commands succeeded
nobtree: commands succeeded
coverage: commands succeeded
docs: commands succeeded
congratulations :)
```

3.5 Contributing to `zope.annotation`

3.5.1 Submitting a Bug Report

`zope.annotation` tracks its bugs on Github:

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

Please submit bug reports and feature requests there.

3.5.2 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.annotation/pulls

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

```
$ bzr push lp:~tseaver/zope.annotation/cool_feature
```

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

Python Module Index

z

Index

Symbols

A

F

G

I

K

Z