zope.catalog Documentation

Release 4.0

Zope Foundation and Contributors

May 09, 2017

Contents

Contents:

CHAPTER 1

Using `zope.catalog`

Basic Usage

Let's look at an example:

```
>>> from zope.catalog.catalog import Catalog
>>> cat = Catalog()
```

We can add catalog indexes to catalogs. A catalog index is, among other things, an attribute index. It indexes attributes of objects. To see how this works, we'll create a demonstration attribute index. Our attribute index will simply keep track of objects that have a given attribute value. The *catalog* package provides an attribute-index mix-in class that is meant to work with a base indexing class. First, we'll write the base index class:

```
>>> import persistent, BTrees.OOBTree, BTrees.IFBTree, BTrees.IOBTree
>>> import zope.interface, zope.index.interfaces

>>> @zope.interface.implementer(
...         zope.index.interfaces.IInjection,
...         zope.index.interfaces.IIndexSearch,
...         zope.index.interfaces.IIndexSort,
...         )
... class BaseIndex(persistent.Persistent):
...
...     def clear(self):
...         self.forward = BTrees.OOBTree.OOBTree()
...         self.backward = BTrees.IOBTree.IOBTree()
...
...     __init__ = clear
...
...     def index_doc(self, docid, value):
...         if docid in self.backward:
...             self.unindex_doc(docid)
...         self.backward[docid] = value
...
```

```
...         set = self.forward.get(value)
...         if set is None:
...             set = BTrees.IFBTree.IFTreeSet()
...             self.forward[value] = set
...         set.insert(docid)
...
...     def unindex_doc(self, docid):
...         value = self.backward.get(docid)
...         if value is None:
...             return
...         self.forward[value].remove(docid)
...         del self.backward[docid]
...
...     def apply(self, value):
...         set = self.forward.get(value)
...         if set is None:
...             set = BTrees.IFBTree.IFTreeSet()
...         return set
...
...     def sort(self, docids, limit=None, reverse=False):
...         key_func = lambda x: self.backward.get(x, -1)
...         for i, docid in enumerate(
...                 sorted(docids, key=key_func, reverse=reverse)):
...             yield docid
...             if limit and i >= (limit - 1):
...                 break
```

The class implements *IInjection* to allow values to be indexed and unindexed and *IIndexSearch* to support searching via the *apply* method.

Now, we can use the AttributeIndex mixin to make this an attribute index:

```
>>> import zope.catalog.attribute
>>> import zope.catalog.interfaces
>>> import zope.container.contained

>>> @zope.interface.implementer(zope.catalog.interfaces.ICatalogIndex)
... class Index(zope.catalog.attribute.AttributeIndex,
...             BaseIndex,
...             zope.container.contained.Contained,
...             ):
...     pass
```

Unfortunately, because of the way we currently handle containment constraints, we have to provide *ICatalogIndex*, which extends *IContained*. We subclass *Contained* to get an implementation for *IContained*.

Now let's add some of these indexes to our catalog. Let's create some indexes. First we'll define some interfaces providing data to index:

```
>>> class IFavoriteColor(zope.interface.Interface):
...     color = zope.interface.Attribute("Favorite color")

>>> class IPerson(zope.interface.Interface):
...     def age():
...         """Return the person's age, in years"""
```

We'll create color and age indexes:

```
>>> cat['color'] = Index('color', IFavoriteColor)
>>> cat['age'] = Index('age', IPerson, True)
>>> cat['size'] = Index('sz')
```

The indexes are created with:

- the name of the of the attribute to index
- the interface defining the attribute, and
- a flag indicating whether the attribute should be called, which defaults to false.

If an interface is provided, then we'll only be able to index an object if it can be adapted to the interface, otherwise, we'll simply try to get the attribute from the object. If the attribute isn't present, then we'll ignore the object.

Now, let's create some objects and index them:

```
>>> @zope.interface.implementer(IPerson)
... class Person:
...     def __init__(self, age):
...         self._age = age
...     def age(self):
...         return self._age

>>> @zope.interface.implementer(IFavoriteColor)
... class Discriminating:
...     def __init__(self, color):
...         self.color = color

>>> class DiscriminatingPerson(Discriminating, Person):
...     def __init__(self, age, color):
...         Discriminating.__init__(self, color)
...         Person.__init__(self, age)

>>> class Whatever:
...     def __init__(self, **kw): #**
...         self.__dict__.update(kw)

>>> o1 = Person(10)
>>> o2 = DiscriminatingPerson(20, 'red')
>>> o3 = Discriminating('blue')
>>> o4 = Whatever(a=10, c='blue', sz=5)
>>> o5 = Whatever(a=20, c='red', sz=6)
>>> o6 = DiscriminatingPerson(10, 'blue')

>>> cat.index_doc(1, o1)
>>> cat.index_doc(2, o2)
>>> cat.index_doc(3, o3)
>>> cat.index_doc(4, o4)
>>> cat.index_doc(5, o5)
>>> cat.index_doc(6, o6)
```

We search by providing query mapping objects that have a key for every index we want to use:

```
>>> list(cat.apply({'age': 10}))
[1, 6]
>>> list(cat.apply({'age': 10, 'color': 'blue'}))
[6]
>>> list(cat.apply({'age': 10, 'color': 'blue', 'size': 5}))
[]
```

```
>>> list(cat.apply({'size': 5}))
[4]
```

We can unindex objects:

```
>>> cat.unindex_doc(4)
>>> list(cat.apply({'size': 5}))
[]
```

and reindex objects:

```
>>> o5.sz = 5
>>> cat.index_doc(5, o5)
>>> list(cat.apply({'size': 5}))
[5]
```

If we clear the catalog, we'll clear all of the indexes:

```
>>> cat.clear()
>>> [len(index.forward) for index in cat.values()]
[0, 0, 0]
```

Note that you don't have to use the catalog's search methods. You can access its indexes directly, since the catalog is a mapping:

```
>>> [(name, cat[name].field_name) for name in cat]
[(u'age', 'age'), (u'color', 'color'), (u'size', 'sz')]
```

Catalogs work with int-id utilities, which are responsible for maintaining id <-> object mappings. To see how this works, we'll create a utility to work with our catalog:

```
>>> import zope.intid.interfaces
>>> @zope.interface.implementer(zope.intid.interfaces.IIntIds)
... class Ids:
...     def __init__(self, data):
...         self.data = data
...     def getObject(self, id):
...         return self.data[id]
...     def __iter__(self):
...         return iter(self.data)
>>> ids = Ids({1: o1, 2: o2, 3: o3, 4: o4, 5: o5, 6: o6})

>>> from zope.component import provideUtility
>>> provideUtility(ids, zope.intid.interfaces.IIntIds)
```

With this utility in place, catalogs can recompute indexes:

```
>>> cat.updateIndex(cat['size'])
>>> list(cat.apply({'size': 5}))
[4, 5]
```

Of course, that only updates *that* index:

```
>>> list(cat.apply({'age': 10}))
[]
```

We can update all of the indexes:

```
>>> cat.updateIndexes()
>>> list(cat.apply({'age': 10}))
[1, 6]
>>> list(cat.apply({'color': 'red'}))
[2]
```

There's an alternate search interface that returns "result sets". Result sets provide access to objects, rather than object ids:

```
>>> result = cat.searchResults(size=5)
>>> len(result)
2
>>> list(result) == [o4, o5]
True
```

The searchResults method also provides a way to sort, limit and reverse results.

When not using sorting, limiting and reversing are done by simple slicing and list reversing.

```
>>> list(cat.searchResults(size=5, _reverse=True)) == [o5, o4]
True

>>> list(cat.searchResults(size=5, _limit=1)) == [o4]
True

>>> list(cat.searchResults(size=5, _limit=1, _reverse=True)) == [o5]
True
```

However, when using sorting by index, the limit and reverse parameters are passed to the index `sort` method so it can do it efficiently.

Let's index more objects to work with:

```
>>> o7 = DiscriminatingPerson(7, 'blue')
>>> o8 = DiscriminatingPerson(3, 'blue')
>>> o9 = DiscriminatingPerson(14, 'blue')
>>> o10 = DiscriminatingPerson(1, 'blue')
>>> ids.data.update({7: o7, 8: o8, 9: o9, 10: o10})
>>> cat.index_doc(7, o7)
>>> cat.index_doc(8, o8)
>>> cat.index_doc(9, o9)
>>> cat.index_doc(10, o10)
```

Now we can search all people who like blue, ordered by age:

```
>>> results = list(cat.searchResults(color='blue', _sort_index='age'))
>>> results == [o3, o10, o8, o7, o6, o9]
True

>>> results = list(cat.searchResults(color='blue', _sort_index='age', _limit=3))
>>> results == [o3, o10, o8]
True

>>> results = list(cat.searchResults(color='blue', _sort_index='age', _reverse=True))
>>> results == [o9, o6, o7, o8, o10, o3]
True

>>> results = list(cat.searchResults(color='blue', _sort_index='age', _reverse=True, _
↪limit=4))
```

```
>>> results == [o9, o6, o7, o8]
True
```

The index example we looked at didn't provide document scores. Simple indexes normally don't, but more complex indexes might give results scores, according to how closely a document matches a query. Let's create a new index, a "keyword index" that indexes sequences of values:

```
>>> @zope.interface.implementer(
...         zope.index.interfaces.IInjection,
...         zope.index.interfaces.IIndexSearch,
...         )
... class BaseKeywordIndex(persistent.Persistent):
...
...     def clear(self):
...         self.forward = BTrees.OOBTree.OOBTree()
...         self.backward = BTrees.IOBTree.IOBTree()
...
...     __init__ = clear
...
...     def index_doc(self, docid, values):
...         if docid in self.backward:
...             self.unindex_doc(docid)
...         self.backward[docid] = values
...
...         for value in values:
...             set = self.forward.get(value)
...             if set is None:
...                 set = BTrees.IFBTree.IFTreeSet()
...                 self.forward[value] = set
...             set.insert(docid)
...
...     def unindex_doc(self, docid):
...         values = self.backward.get(docid)
...         if values is None:
...             return
...         for value in values:
...             self.forward[value].remove(docid)
...         del self.backward[docid]
...
...     def apply(self, values):
...         result = BTrees.IFBTree.IFBucket()
...         for value in values:
...             set = self.forward.get(value)
...             if set is not None:
...                 _, result = BTrees.IFBTree.weightedUnion(result, set)
...         return result

>>> @zope.interface.implementer(zope.catalog.interfaces.ICatalogIndex)
... class KeywordIndex(zope.catalog.attribute.AttributeIndex,
...                    BaseKeywordIndex,
...                    zope.container.contained.Contained,
...                    ):
...     pass
```

Now, we'll add a hobbies index:

```
>>> cat['hobbies'] = KeywordIndex('hobbies')
>>> o1.hobbies = 'camping', 'music'
```

```
>>> o2.hobbies = 'hacking', 'sailing'
>>> o3.hobbies = 'music', 'camping', 'sailing'
>>> o6.hobbies = 'cooking', 'dancing'
>>> cat.updateIndexes()
```

When we apply the catalog:

```
>>> cat.apply({'hobbies': ['music', 'camping', 'sailing']})
BTrees.IFBTree.IFBucket([(1, 2.0), (2, 1.0), (3, 3.0)])
```

We found objects 1-3, because they each contained at least some of the words in the query. The scores represent the number of words that matched. If we also include age:

```
>>> cat.apply({'hobbies': ['music', 'camping', 'sailing'], 'age': 10})
BTrees.IFBTree.IFBucket([(1, 3.0)])
```

The score increased because we used an additional index. If an index doesn't provide scores, scores of 1.0 are assumed.

Additional Topics

Automatic indexing with events

In order to automatically keep the catalog up-to-date any objects that are added to a intid utility are indexed automatically. Also when an object gets modified it is reindexed by listening to IObjectModified events.

Let us create a fake catalog to demonstrate this behaviour. We only need to implement the index_doc method for this test.

```
>>> from zope.catalog.interfaces import ICatalog
>>> from zope import interface, component
>>> @interface.implementer(ICatalog)
... class FakeCatalog(object):
...     indexed = []
...     def index_doc(self, docid, obj):
...         self.indexed.append((docid, obj))
>>> cat = FakeCatalog()
>>> component.provideUtility(cat)
```

We also need an intid util and a keyreference adapter.

```
>>> from zope.intid import IntIds
>>> from zope.intid.interfaces import IIntIds
>>> intids = IntIds()
>>> component.provideUtility(intids, IIntIds)
>>> from zope.keyreference.testing import SimpleKeyReference
>>> component.provideAdapter(SimpleKeyReference)

>>> from  zope.container.contained import Contained
>>> class Dummy(Contained):
...     def __init__(self, name):
...         self.__name__ = name
...     def __repr__(self):
...         return '<Dummy %r>' % self.__name__
```

We have a subscriber to IIntidAddedEvent.

```
>>> from zope.catalog import catalog
>>> from zope.intid.interfaces import IntIdAddedEvent
>>> d1 = Dummy(u'one')
>>> id1 = intids.register(d1)
>>> catalog.indexDocSubscriber(IntIdAddedEvent(d1, None))
```

Now we have indexed the object.

```
>>> cat.indexed.pop()
(..., <Dummy u'one'>)
```

When an object is modified an objectmodified event should be fired by the application. Here is the handler for such an event.

```
>>> from zope.lifecycleevent import ObjectModifiedEvent
>>> catalog.reindexDocSubscriber(ObjectModifiedEvent(d1))
>>> len(cat.indexed)
1
>>> cat.indexed.pop()
(..., <Dummy u'one'>)
```

Preventing automatic indexing

Sometimes it is not accurate to automatically index an object. For example when a lot of indexes are in the catalog and only specific indexes needs to be updated. There are marker interfaces to achieve this.

```
>>> from zope.catalog.interfaces import INoAutoIndex
```

If an object provides this interface it is not automatically indexed.

```
>>> interface.alsoProvides(d1, INoAutoIndex)
>>> catalog.indexDocSubscriber(IntIdAddedEvent(d1, None))
>>> len(cat.indexed)
0

>>> from zope.catalog.interfaces import INoAutoReindex
```

If an object provides this interface it is not automatically reindexed.

```
>>> interface.alsoProvides(d1, INoAutoReindex)
>>> catalog.reindexDocSubscriber(ObjectModifiedEvent(d1))
>>> len(cat.indexed)
0
```

CHAPTER 2

`zope.catalog` API

Interfaces

CHAPTER 3

Hacking on `zope.catalog`

Getting the Code

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

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

You can get a read-only checkout from there:

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

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

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

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

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

You can branch the trunk from there using Bazaar:

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

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.catalog
```

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

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

Running the tests

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

```
$ /tmp/hack-zope.catalog/bin/python setup.py test
running test
.................
----------------------------------------------------------------------
Ran 17 tests in 0.000s

OK
```

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

```
$ /tmp/hack-zope.catalog/bin/easy_install nose
...
$ /tmp/hack-zope.catalog/bin/nosetests
.................
----------------------------------------------------------------------
Ran 17 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.catalog/bin/easy_install nose coverage
...
$ /tmp/hack-zope.catalog/bin/nosetests --with coverage
running nosetests
...................
Name                          Stmts   Miss  Cover   Missing
-----------------------------------------------------------
zope/catalog.py                   0      0   100%
zope/catalog/attribute.py        31      0   100%
zope/catalog/catalog.py         125      0   100%
zope/catalog/field.py            10      0   100%
zope/catalog/interfaces.py       22      0   100%
zope/catalog/keyword.py          13      0   100%
zope/catalog/text.py             15      0   100%
-----------------------------------------------------------
TOTAL                           216     16   100%
----------------------------------------------------------------------
Ran 19 tests in 0.554s

OK
```

Building the documentation

`zope.catalog` 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.catalog/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
===============
  117 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.catalog` 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/zope.catalog/.'
...
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.catalog configures the following tox environments via its tox.ini file:

- The py26, py27, py33, py34, and pypy environments builds a virtualenv with the appropriate interpreter, installs zope.catalog and dependencies, and runs the tests via python setup.py test -q.
- The coverage environment builds a virtualenv with python2.6, installs zope.catalog, installs nose and coverage, and runs nosetests with statement coverage.
- The docs environment builds a virtualenv with python2.6, installs zope.catalog, 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 17 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
===============
 117 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
coverage: commands succeeded
docs: commands succeeded
congratulations :)
```

Contributing to zope.catalog

Submitting a Bug Report

zope.catalog tracks its bugs on Github:

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