lazr.lifecycle Documentation

Release 1.2.1

LAZR Developers <lazr-developers@lists.launchpad.net>

Nov 03, 2021

CONTENTS

This package defines three "lifecycle" events that notify about object creation, modification and deletion. The events include information about the user responsible for the changes.

The modification event also includes information about the state of the object before the changes.

The module also contains Snapshot support to save the state of an object for notification, and to compute deltas between version of objects.

CHAPTER

ONE

OTHER DOCUMENTS

1.1 LAZR Lifecycle Events

lazr.lifecycle defines common lifecycle events. They are extensions of the base Z3 lifecycle events.

```
>>> from lazr.lifecycle.interfaces import (
...     IObjectCreatedEvent, IObjectModifiedEvent,
...     IObjectDeletedEvent)
>>> from lazr.lifecycle.event import (
...     ObjectCreatedEvent, ObjectDeletedEvent, ObjectModifiedEvent)
```

```
>>> from zope.interface.verify import verifyObject
```

```
>>> class Fnord:
...     "A Fnort sighting."""
...
...     def __init__(self, who, where):
...         self.who = who
...         self.where = where
```

The module defines three lifecycle events:

- ObjectCreatedEvent - Used when an object has been created.
- ObjectModifiedEvent - Used then an object has been modified, it provides meta data explaining what was modified.
- ObjectDeletedEvent - Used when the object was deleted.

All events expose the user responsible for the change in the user attribute. By default, if not specified when constructing the event, this will be the principal associated with the interaction. (Only one principal is currently supported)

```
>>> from zope.security.management import (
...     newInteraction, endInteraction)
```

```
>>> class MyParticipation:
...     def __init__(self, who):
...         self.principal = who
...         self.interaction = None
```

```
>>> newInteraction(MyParticipation('the user'))
```

```
>>> fnord = Fnord('me', 'the_bridge')
>>> created_event = ObjectCreatedEvent(fnord)
>>> verifyObject(IObjectCreatedEvent, created_event)
True
```

```
>>> print(created_event.user)
the user
```

The object attribute contains the object that was created:

```
>>> created_event.object is fnord
True
```

The ObjectModifiedEvent holds the names of the modified fields in the 'edited_fields' attribute and the state of the object before the modifications in 'object_before_modification' attribute. (See snapshot.txt for a way to manage that easily).

```
>>> snapshot = Fnord('me', 'the_bridge')
>>> fnord.who = 'someone else'
```

```
>>> modified_event = ObjectModifiedEvent(fnord, snapshot, ['who'])
```

```
>>> verifyObject(IObjectModifiedEvent, modified_event)
True
```

```
>>> modified_event.edited_fields
['who']
>>> modified_event.object is fnord
True
>>> print(modified_event.object_before_modification.who)
me
>>> print(modified_event.user)
the user
```

The ObjectDeletedEvent is used to broadcast the deletion of the object.

```
>>> deleted_event = ObjectDeletedEvent(fnord, user='the_censor')
```

```
>>> verifyObject(IObjectDeletedEvent, deleted_event)
True
```

```
>>> deleted_event.object is fnord
True
>>> print(deleted_event.user)
the_censor
```

1.2 Creating Object Deltas

A common workflow surrounding life-cycle events involves computing the deltas between two objects instances. This is normally for notification emails in order to identify what has been changed, so the appropriate information is sent out in the email.

In order to test that the deltas, we need a simple object to test.

```
>>> class TestObj(object):
...   def __init__(self, vars):
...     self.__dict__.update(vars)
>>> old_obj = TestObj({'foo':42, 'bar':'hello', 'baz':[1,2,3], 'wiz':1})
>>> new_obj = TestObj({'foo':42, 'bar':'world', 'baz':[2,3,4], 'wiz':2})
>>> old_obj.foo
42
>>> old_obj.bar
'hello'
```

And here is a simple helper function that we'll use to print out dicts in sorted order (as pprint doesn't assure this until 2.5).

```
>>> def display_delta(to_display, indent=0):
...   for key in sorted(to_display.keys()):
...     value = to_display[key]
...     if isinstance(value, dict):
...       print("%s: %s => %s" % (key, repr(value['old']),
...                                repr(value['new'])))
...     else:
...       print("%s: %s" % (key, repr(to_display[key])))
```

The ObjectDelta instance is constructed with the old object, and the new object instances.

```
>>> from lazr.lifecycle.objectdelta import ObjectDelta
>>> delta = ObjectDelta(old_obj, new_obj)
```

The simplest case is are recording of new values. The changes property of the delta object just shows the field name and the new value.

```
>>> delta.recordNewValues(['foo', 'bar', 'wiz'])
>>> display_delta(delta.changes)
bar: 'world'
wiz: 2
```

An alternative to just returning the new values is to return both the old and the new values for a field value.

```
>>> delta = ObjectDelta(old_obj, new_obj)
>>> delta.recordNewAndOld(['foo', 'bar', 'wiz'])
>>> display_delta(delta.changes)
bar: 'hello' => 'world'
wiz: 1 => 2
```

In some cases, a list is needed to be checked for additions and removals. Examples for this would be associated bugs, specs or brances with the specified object.

```
>>> delta = ObjectDelta(old_obj, new_obj)
>>> delta.recordListAddedAndRemoved('baz', 'added', 'removed')
>>> display_delta(delta.changes)
added: [4]
removed: [1]
```

Now the normal usage is to do some combination of the above, and pass the resulting map through to the constructor for a real delta object.

```
>>> class TestDelta:
...   def __init__(self, foo=None, bar=None, wiz=None,
...                baz_added=None, baz_removed=None):
...     self.foo = foo
...     self.bar = bar
...     self.baz_added = baz_added
...     self.baz_removed= baz_removed
...     self.wiz = wiz
```

```
>>> obj_delta = ObjectDelta(old_obj, new_obj)
>>> obj_delta.recordNewValues(['foo', 'bar'])
>>> obj_delta.recordNewAndOld(['wiz'])
>>> obj_delta.recordListAddedAndRemoved('baz', 'baz_added',
...                                     'baz_removed')
>>> delta = TestDelta(**obj_delta.changes)
>>> delta.foo
>>> delta.bar
'world'
>>> delta.baz_added
[4]
>>> delta.baz_removed
[1]
>>> display_delta(delta.wiz)
new: 2
old: 1
```

1.3 Get a Snapshot of an Object

The lazr.lifecycle.event.ObjectModifiedEvent has an attribute giving the initial state of the object before the modifications. The Snapshot class can be used to easily represent such states:

```
>>> from zope.interface import Attribute, Interface, implementer
>>> from zope.schema import List, TextLine, Text
>>> from lazr.lifecycle.snapshot import doNotSnapshot, Snapshot
```

```
>>> class IFoo(Interface):
...     title = TextLine(title=u'My Title')
...     description = Text(title=u'Description')
...     remotes = List(title=u'remotes')
...     totals = Attribute('totals')
...     ignore_me = doNotSnapshot(Attribute('ignored attribute'))
```

```
>>> @implementer(IFoo)
... class Foo:
...     remotes = ["OK"]
...
...     @property
...     def totals(self):
...         return "NOT"
>>> foo = Foo()
>>> foo.title = 'Some Title'
>>> foo.description = 'bla bla bla'
```

A snapshot can be created by specifying the names of the attribute you want to snapshot:

```
>>> snapshot = Snapshot(foo, names=['title'])
```

Only the given attributes will be assigned to the snapshot:

```
>>> snapshot.title == foo.title
True
>>> hasattr(snapshot, 'description')
False
```

The snapshot won't provide the same interface as foo, though:

```
>>> IFoo.providedBy(snapshot)
False
```

If we want the snapshot to provide some interface, we have to specify that explicitly:

```
>>> snapshot = Snapshot(foo, names=['title'], providing=IFoo)
>>> snapshot.title == foo.title
True
>>> hasattr(snapshot, 'description')
False
>>> IFoo.providedBy(snapshot)
True
```

The API requires you to specify either 'names' or 'providing':

```
>>> snapshot = Snapshot(foo)
Traceback (most recent call last):
...
SnapshotCreationError
```

If no names argument is supplied, the snapshot will contain all IFields of the specified interface(s).

```
>>> snapshot = Snapshot(foo, providing=IFoo)
>>> snapshot.title == foo.title
True
>>> hasattr(snapshot, 'description')
True
```

Totals is not a Fields so isn't copied over.

```
>>> hasattr(snapshot, 'totals')
False
>>> hasattr(snapshot, 'remotes')
True
>>> snapshot.remotes == ["OK"]
True
```

Fields can be explicitly ignored if they provide the IDoNotSnapshot interface.

```
>>> hasattr(snapshot, 'ignore_me')
False
```

We can also give more than one interface to provide as an iterable. If we don't specify any names, all the names in the given interfaces will be copied:

```
>>> from zope.interface import providedBy
>>> snapshot = Snapshot(foo, providing=providedBy(foo))
>>> snapshot.title == foo.title
True
>>> snapshot.description == foo.description
True
>>> IFoo.providedBy(snapshot)
True
```

```
>>> class IBar(Interface):
...     name = Text(title=u'Name')
```

```
>>> foo.name = "barbie"
```

```
>>> snapshot = Snapshot(foo, providing=[IFoo, IBar])
>>> IFoo.providedBy(snapshot)
True
>>> IBar.providedBy(snapshot)
True
```

```
>>> snapshot.title == foo.title
True
>>> snapshot.name == "barbie"
True
>>> snapshot.description == foo.description
True
>>> IFoo.providedBy(snapshot)
True
>>> IBar.providedBy(snapshot)
True
```

1.4 ISnapshotValueFactory

For some fields, assigning the existing value to the snapshot object isn't appropriate. For these case, one can provide a factory registered as an adapter for the value to ISnapshotValueFactory. The result of the adaptation lookup will be stored in the snapshot attribute.

```
>>> from zope.interface import implementer, Interface
>>> from zope.component import adapter, getSiteManager
```

```
>>> class IIterable(Interface):
...     """Marker for a value that needs a special snapshot."""
```

```
>>> @implementer(IIterable)
... class EvenOrOddIterable:
...     """An object that will be snapshotted specially."""
...     even = True
...     max = 10
...
...     def __iter__(self):
...         for i in range(self.max):
...             if i % 2  == 0 and self.even:
...                 yield i
...             elif i % 2  == 1 and not self.even:
...                 yield i
...             else:
...                 continue
```

```
>>> from lazr.lifecycle.interfaces import ISnapshotValueFactory
>>> @implementer(ISnapshotValueFactory)
... @adapter(IIterable)
... def snapshot_iterable(value):
...     return list(value)
>>> getSiteManager().registerAdapter(snapshot_iterable)
```

```
>>> foo = Foo()
>>> foo.title = 'Even'
>>> foo.description = 'Generates even number below 10.'
>>> foo.remotes = EvenOrOddIterable()
```

```
>>> snapshot = Snapshot(foo, providing=IFoo)
>>> snapshot.remotes == list(foo.remotes)
True
```

```
>>> getSiteManager().unregisterAdapter(snapshot_iterable)
True
```

1.5 NEWS for lazr.lifecycle

1.5.1 1.2.1 (2021-09-13)

- Adjust versioning strategy to avoid importing pkg_resources, which is slow in large environments.

1.5.2 1.2 (2019-11-04)

- Import IObjectEvent from zope.interface rather than zope.component.
- Add ObjectModifiedEvent.descriptions property, for compatibility with zope.lifecycleevent >= 4.2.0.
- Switch from buildout to tox.
- Add Python 3 support.

1.5.3 1.1 (2009-12-03)

- Add IDoNotSnapshot and doNotSnapshot to allow the exclusion of certain fields.

1.5.4 1.0 (2009-08-31)

- Remove build dependencies on bzr and egg_info
- remove sys.path hack in setup.py for __version__

1.5.5 0.1 (2009-03-24)

- Initial release