zope.configuration Documentation

Release 4.0

Zope Foundation Contributors

Sep 27, 2017

Contents

Contents:

CHAPTER 1

Zope configuration system

The zope configuration system provides an extensible system for supporting variouse kinds of configurations.

It is based on the idea of configuration directives. Users of the configuration system provide configuration directives in some language that express configuration choices. The intent is that the language be pluggable. An XML language is provided by default.

Configuration is performed in three stages. In the first stage, directives are processed to compute configuration actions. Configuration actions consist of:

- A discriminator
- A callable
- Positional arguments
- Keyword arguments

The actions are essentially delayed function calls. Two or more actions conflict if they have the same discriminator. The configuration system has rules for resolving conflicts. If conflicts cannot be resolved, an error will result. Conflict resolution typically discards all but one of the conflicting actions, so that the remaining action of the originally-conflicting actions no longer conflicts. Non-conflicting actions are executed in the order that they were created by passing the positional and non-positional arguments to the action callable.

The system is extensible. There is a meta-configuration language for defining configuration directives. A directive is defined by providing meta data about the directive and handler code to process the directive. There are four kinds of directives:

- Simple directives compute configuration actions. Their handlers are typically functions that take a context and zero or more keyword arguments and return a sequence of configuration actions.

  To learn how to create simple directives, see *tests/simple.py*.

- Grouping directives collect information to be used by nested directives. They are called with a context object which they adapt to some interface that extends IConfigurationContext.

  To learn how to create grouping directives, look at the documentation in zopeconfigure.py, which provides the implementation of the zope *configure* directive.

  Other directives can be nested in grouping directives.

To learn how to implement nested directives, look at the documentation in the "Creating Nested Directives" section below.

- Complex directives are directives that have subdirectives. Subdirectives have handlers that are simply methods of complex directives. Complex diretives are handled by factories, typically classes, that create objects that have methods for handling subdirectives. These objects also have __call__ methods that are called when processing of subdirectives is finished.

  Complex directives only exist to support old directive handlers. They will probably be deprecated in the future.

- Subdirectives are nested in complex directives. They are like simple directives except that they hane handlers that are complex directive methods.

  Subdirectives, like complex directives only exist to support old directive handlers. They will probably be deprecated in the future.

Using the configuration machinery programatically

An extended example:

```
>>> from zope.configuration.config import ConfigurationMachine
>>> from zope.configuration.config import metans
>>> machine = ConfigurationMachine()
>>> ns = "http://www.zope.org/testing"
```

Register some test directives:

Start with a grouping directive that sets a package:

```
>>> machine((metans, "groupingDirective"),
...         name="package", namespace=ns,
...         schema="zope.configuration.tests.directives.IPackaged",
...         handler="zope.configuration.tests.directives.Packaged",
...         )
```

Now we can set the package:

```
>>> machine.begin((ns, "package"),
...               package="zope.configuration.tests.directives",
...               )
```

Which makes it easier to define the other directives:

First, define some simple directives:

```
>>> machine((metans, "directive"),
...         namespace=ns, name="simple",
...         schema=".ISimple", handler=".simple")

>>> machine((metans, "directive"),
...         namespace=ns, name="newsimple",
...         schema=".ISimple", handler=".newsimple")
```

and try them out:

```
>>> machine((ns, "simple"), "first", a=u"aa", c=u"cc")
>>> machine((ns, "newsimple"), "second", a=u"naa", c=u"ncc", b=u"nbb")
```

```
>>> from pprint import PrettyPrinter
>>> pprint = PrettyPrinter(width=50).pprint

>>> pprint(machine.actions)
[{'args': (u'aa', u'xxx', 'cc'),
  'callable': f,
  'discriminator': ('simple',
                    u'aa',
                    u'xxx',
                    'cc'),
  'includepath': (),
  'info': 'first',
  'kw': {},
  'order': 0},
 {'args': (u'naa', u'nbb', 'ncc'),
  'callable': f,
  'discriminator': ('newsimple',
                    u'naa',
                    u'nbb',
                    'ncc'),
  'includepath': (),
  'info': 'second',
  'kw': {},
  'order': 0}]
```

Define and try a simple directive that uses a component:

```
>>> machine((metans, "directive"),
...         namespace=ns, name="factory",
...         schema=".IFactory", handler=".factory")


>>> machine((ns, "factory"), factory=u".f")
>>> pprint(machine.actions[-1:])
[{'args': (),
  'callable': f,
  'discriminator': ('factory', 1, 2),
  'includepath': (),
  'info': None,
  'kw': {},
  'order': 0}]
```

Define and try a complex directive:

```
>>> machine.begin((metans, "complexDirective"),
...               namespace=ns, name="testc",
...               schema=".ISimple", handler=".Complex")

>>> machine((metans, "subdirective"),
...         name="factory", schema=".IFactory")

>>> machine.end()

>>> machine.begin((ns, "testc"), None, "third", a=u'ca', c='cc')
>>> machine((ns, "factory"), "fourth", factory=".f")
```

Note that we can't call a complex method unless there is a directive for it:

```
>>> machine((ns, "factory2"), factory=".f")
Traceback (most recent call last):
...
ConfigurationError: ('Invalid directive', 'factory2')


>>> machine.end()
>>> pprint(machine.actions)
[{'args': (u'aa', u'xxx', 'cc'),
  'callable': f,
  'discriminator': ('simple',
                    u'aa',
                    u'xxx',
                    'cc'),
  'includepath': (),
  'info': 'first',
  'kw': {},
  'order': 0},
 {'args': (u'naa', u'nbb', 'ncc'),
  'callable': f,
  'discriminator': ('newsimple',
                    u'naa',
                    u'nbb',
                    'ncc'),
  'includepath': (),
  'info': 'second',
  'kw': {},
  'order': 0},
 {'args': (),
  'callable': f,
  'discriminator': ('factory', 1, 2),
  'includepath': (),
  'info': None,
  'kw': {},
  'order': 0},
 {'args': (),
  'callable': None,
  'discriminator': 'Complex.__init__',
  'includepath': (),
  'info': 'third',
  'kw': {},
  'order': 0},
 {'args': (u'ca',),
  'callable': f,
  'discriminator': ('Complex.factory', 1, 2),
  'includepath': (),
  'info': 'fourth',
  'kw': {},
  'order': 0},
 {'args': (u'xxx', 'cc'),
  'callable': f,
  'discriminator': ('Complex', 1, 2),
  'includepath': (),
  'info': 'third',
  'kw': {},
  'order': 0}]
```

Done with the package

```
>>> machine.end()
```

Verify that we can use a simple directive outside of the package:

```
>>> machine((ns, "simple"), a=u"oaa", c=u"occ", b=u"obb")
```

But we can't use the factory directive, because it's only valid inside a package directive:

```
>>> machine((ns, "factory"), factory=u".F")
Traceback (most recent call last):
...
ConfigurationError: ('Invalid value for', 'factory',""" \
   """ "Can't use leading dots in dotted names, no package has been set.")

>>> pprint(machine.actions)
[{'args': (u'aa', u'xxx', 'cc'),
  'callable': f,
  'discriminator': ('simple',
                    u'aa',
                    u'xxx',
                    'cc'),
  'includepath': (),
  'info': 'first',
  'kw': {},
  'order': 0},
 {'args': (u'naa', u'nbb', 'ncc'),
  'callable': f,
  'discriminator': ('newsimple',
                    u'naa',
                    u'nbb',
                    'ncc'),
  'includepath': (),
  'info': 'second',
  'kw': {},
  'order': 0},
 {'args': (),
  'callable': f,
  'discriminator': ('factory', 1, 2),
  'includepath': (),
  'info': None,
  'kw': {},
  'order': 0},
 {'args': (),
  'callable': None,
  'discriminator': 'Complex.__init__',
  'includepath': (),
  'info': 'third',
  'kw': {},
  'order': 0},
 {'args': (u'ca',),
  'callable': f,
  'discriminator': ('Complex.factory', 1, 2),
  'includepath': (),
  'info': 'fourth',
  'kw': {},
  'order': 0},
 {'args': (u'xxx', 'cc'),
  'callable': f,
```

```
  'discriminator': ('Complex', 1, 2),
  'includepath': (),
  'info': 'third',
  'kw': {},
  'order': 0},
 {'args': (u'oaa', u'obb', 'occ'),
  'callable': f,
  'discriminator': ('simple',
                    u'oaa',
                    u'obb',
                    'occ'),
  'includepath': (),
  'info': None,
  'kw': {},
  'order': 0}]
```

Overriding Included Configuration

When we have conflicting directives, we can resolve them if one of the conflicting directives was from a file that included all of the others. The problem with this is that this requires that all of the overriding directives be in one file, typically the top-most including file. This isn't very convenient. Fortunately, we can overcome this with the includeOverrides directive. Let's look at an example to see how this works.

Look at the file `bar.zcml` (in `zope/configuration/tests/samplepackage`):

- It includes `bar1.zcml` and `bar2.zcml`.
- `bar1.zcml` includes `configure.zcml` and has a `foo` directive.
- `bar2.zcml` includes `bar21.zcml`, and has a `foo` directive that conflicts with one in `bar1.zcml`.
- `bar2.zcml` also overrides a foo directive in `bar21.zcml`.
- `bar21.zcml` has a `foo` directive that conflicts with one in in `configure.zcml`. Whew!

Let's see what happens when we try to process `bar.zcml`.

```
>>> import os
>>> from zope.configuration.config import ConfigurationMachine
>>> from zope.configuration.xmlconfig import include
>>> from zope.configuration.xmlconfig import registerCommonDirectives
>>> context = ConfigurationMachine()
>>> registerCommonDirectives(context)

>>> from zope.configuration.tests import __file__
>>> here = os.path.dirname(__file__)
>>> path = os.path.join(here, "samplepackage", "bar.zcml")
>>> include(context, path)
```

So far so good, let's look at the configuration actions:

```
>>> from zope.configuration.tests.test_xmlconfig import clean_actions
>>> pprint = PrettyPrinter(width=70).pprint
>>> pprint(clean_actions(context.actions))
[{'discriminator': (('x', 'blah'), ('y', 0)),
  'includepath': ['tests/samplepackage/bar.zcml',
                  'tests/samplepackage/bar1.zcml',
                  'tests/samplepackage/configure.zcml'],
```

```
  'info': 'File "tests/samplepackage/configure.zcml", line 12.2-12.29'},
 {'discriminator': (('x', 'blah'), ('y', 1)),
  'includepath': ['tests/samplepackage/bar.zcml',
                  'tests/samplepackage/bar1.zcml'],
  'info': 'File "tests/samplepackage/bar1.zcml", line 5.2-5.24'},
 {'discriminator': (('x', 'blah'), ('y', 0)),
  'includepath': ['tests/samplepackage/bar.zcml',
                  'tests/samplepackage/bar2.zcml',
                  'tests/samplepackage/bar21.zcml'],
  'info': 'File "tests/samplepackage/bar21.zcml", line 3.2-3.24'},
 {'discriminator': (('x', 'blah'), ('y', 2)),
  'includepath': ['tests/samplepackage/bar.zcml',
                  'tests/samplepackage/bar2.zcml',
                  'tests/samplepackage/bar21.zcml'],
  'info': 'File "tests/samplepackage/bar21.zcml", line 4.2-4.24'},
 {'discriminator': (('x', 'blah'), ('y', 2)),
  'includepath': ['tests/samplepackage/bar.zcml',
                  'tests/samplepackage/bar2.zcml'],
  'info': 'File "tests/samplepackage/bar2.zcml", line 5.2-5.24'},
 {'discriminator': (('x', 'blah'), ('y', 1)),
  'includepath': ['tests/samplepackage/bar.zcml',
                  'tests/samplepackage/bar2.zcml'],
  'info': 'File "tests/samplepackage/bar2.zcml", line 6.2-6.24'}]
```

As you can see, there are a number of conflicts (actions with the same discriminator). Some of these can be resolved, but many can't, as we'll find if we try to execuse the actions:

```
>>> from zope.configuration.config import ConfigurationConflictError
>>> from zope.configuration.tests.test_xmlconfig import clean_text_w_paths
>>> try:
...     v = context.execute_actions()
... except ConfigurationConflictError, v:
...     pass
>>> print clean_text_w_paths(str(v))
Conflicting configuration actions
  For: (('x', 'blah'), ('y', 0))
    File "tests/samplepackage/configure.zcml", line 12.2-12.29
        <test:foo x="blah" y="0" />
    File "tests/samplepackage/bar21.zcml", line 3.2-3.24
        <foo x="blah" y="0" />
  For: (('x', 'blah'), ('y', 1))
    File "tests/samplepackage/bar1.zcml", line 5.2-5.24
        <foo x="blah" y="1" />
    File "tests/samplepackage/bar2.zcml", line 6.2-6.24
        <foo x="blah" y="1" />
```

Note that the conflicts for (('x', 'blah'), ('y', 2)) aren't included in the error because they could be resolved.

Let's try this again using includeOverrides. We'll include baro.zcml which includes bar2.zcml as overrides.

```
>>> context = ConfigurationMachine()
>>> registerCommonDirectives(context)
>>> path = os.path.join(here, "samplepackage", "baro.zcml")
>>> include(context, path)
```

Now, if we look at the actions:

```
>>> pprint(clean_actions(context.actions))
[{'discriminator': (('x', 'blah'), ('y', 0)),
  'includepath': ['tests/samplepackage/baro.zcml',
                  'tests/samplepackage/bar1.zcml',
                  'tests/samplepackage/configure.zcml'],
  'info': 'File "tests/samplepackage/configure.zcml", line 12.2-12.29'},
 {'discriminator': (('x', 'blah'), ('y', 1)),
  'includepath': ['tests/samplepackage/baro.zcml',
                  'tests/samplepackage/bar1.zcml'],
  'info': 'File "tests/samplepackage/bar1.zcml", line 5.2-5.24'},
 {'discriminator': (('x', 'blah'), ('y', 0)),
  'includepath': ['tests/samplepackage/baro.zcml'],
  'info': 'File "tests/samplepackage/bar21.zcml", line 3.2-3.24'},
 {'discriminator': (('x', 'blah'), ('y', 2)),
  'includepath': ['tests/samplepackage/baro.zcml'],
  'info': 'File "tests/samplepackage/bar2.zcml", line 5.2-5.24'},
 {'discriminator': (('x', 'blah'), ('y', 1)),
  'includepath': ['tests/samplepackage/baro.zcml'],
  'info': 'File "tests/samplepackage/bar2.zcml", line 6.2-6.24'}]
```

We see that:

- The conflicting actions between bar2.zcml and bar21.zcml have been resolved, and
- The remaining (after conflict resolution) actions from bar2.zcml and bar21.zcml have the includepath that they would have if they were defined in baro.zcml and this override the actions from bar1.zcml and configure.zcml.

We can now execute the actions without problem, since the remaining conflicts are resolvable:

```
>>> context.execute_actions()
```

We should now have three entries in foo.data:

```
>>> from zope.configuration.tests.samplepackage import foo
>>> from zope.configuration.tests.test_xmlconfig import clean_info_path
>>> len(foo.data)
3

>>> data = foo.data.pop(0)
>>> data.args
(('x', 'blah'), ('y', 0))
>>> print clean_info_path(`data.info`)
File "tests/samplepackage/bar21.zcml", line 3.2-3.24

>>> data = foo.data.pop(0)
>>> data.args
(('x', 'blah'), ('y', 2))
>>> print clean_info_path(`data.info`)
File "tests/samplepackage/bar2.zcml", line 5.2-5.24

>>> data = foo.data.pop(0)
>>> data.args
(('x', 'blah'), ('y', 1))
>>> print clean_info_path(`data.info`)
File "tests/samplepackage/bar2.zcml", line 6.2-6.24
```

We expect the exact same results when using includeOverrides with the `files` argument instead of the `file` argument. The baro2.zcml file uses the former:

```
>>> context = ConfigurationMachine()
>>> registerCommonDirectives(context)
>>> path = os.path.join(here, "samplepackage", "baro2.zcml")
>>> include(context, path)
```

Actions look like above:

```
>>> pprint(clean_actions(context.actions))
[{'discriminator': (('x', 'blah'), ('y', 0)),
  'includepath': ['tests/samplepackage/baro2.zcml',
                  'tests/samplepackage/bar1.zcml',
                  'tests/samplepackage/configure.zcml'],
  'info': 'File "tests/samplepackage/configure.zcml", line 12.2-12.29'},
 {'discriminator': (('x', 'blah'), ('y', 1)),
  'includepath': ['tests/samplepackage/baro2.zcml',
                  'tests/samplepackage/bar1.zcml'],
  'info': 'File "tests/samplepackage/bar1.zcml", line 5.2-5.24'},
 {'discriminator': (('x', 'blah'), ('y', 0)),
  'includepath': ['tests/samplepackage/baro2.zcml'],
  'info': 'File "tests/samplepackage/bar21.zcml", line 3.2-3.24'},
 {'discriminator': (('x', 'blah'), ('y', 2)),
  'includepath': ['tests/samplepackage/baro2.zcml'],
  'info': 'File "tests/samplepackage/bar2.zcml", line 5.2-5.24'},
 {'discriminator': (('x', 'blah'), ('y', 1)),
  'includepath': ['tests/samplepackage/baro2.zcml'],
  'info': 'File "tests/samplepackage/bar2.zcml", line 6.2-6.24'}]

>>> context.execute_actions()
>>> len(foo.data)
3
>>> del foo.data[:]
```

Making specific directives conditional

There is a `condition` attribute in the "http://namespaces.zope.org/zcml" namespace which is honored on all elements in ZCML. The value of the attribute is an expression which is used to determine if that element and its descendents are used. If the condition is true, processing continues normally, otherwise that element and its descendents are ignored.

Currently the expression is always of the form "have featurename", and it checks for the presence of a `<meta:provides feature="featurename" />`.

Our demonstration uses a trivial registry; each registration consists of a simple id inserted in the global *registry* in this module. We can checked that a registration was made by checking whether the id is present in *registry*.

```
>>> from zope.configuration.tests.conditions import registry
>>> registry
[]
```

We start by loading the example ZCML file, *conditions.zcml*:

```
>>> import zope.configuration.tests
>>> from zope.configuration.xmlconfig import file
>>> context = file("conditions.zcml", zope.configuration.tests)
```

To show that our sample directive works, we see that the unqualified registration was successful:

```
>>> "unqualified.registration" in registry
True
```

When the expression specified with `zcml:condition` evaluates to true, the element it is attached to and all contained elements (not otherwise conditioned) should be processed normally:

```
>>> "direct.true.condition" in registry
True
>>> "nested.true.condition" in registry
True
```

However, when the expression evaluates to false, the conditioned element and all contained elements should be ignored:

```
>>> "direct.false.condition" in registry
False
>>> "nested.false.condition" in registry
False
```

Conditions on container elements affect the conditions in nested elements in a reasonable way. If an "outer" condition is true, nested conditions are processed normally:

```
>>> "true.condition.nested.in.true" in registry
True
>>> "false.condition.nested.in.true" in registry
False
```

If the outer condition is false, inner conditions are not even evaluated, and the nested elements are ignored:

```
>>> "true.condition.nested.in.false" in registry
False
>>> "false.condition.nested.in.false" in registry
False
```

Filtering and Inhibiting Configuration

The `exclude` standard directive is provided for inhibiting unwanted configuration. It is used to exclude processing of configuration files. It is useful when including a configuration that includes some other configuration that you don't want.

It must be used BEFORE including the files to be excluded.

First, let's look at an example. The zope.configuration.tests.excludedemo package has a ZCML configuration that includes some other configuration files.

We'll set a log handler so we can see what's going on:

```
>>> import logging
>>> import logging.handlers
>>> import sys
>>> logger = logging.getLogger('config')
>>> oldlevel = logger.level
>>> logger.setLevel(logging.DEBUG)
>>> handler = logging.handlers.MemoryHandler(10)
>>> logger.addHandler(handler)
```

Now, we'll include the zope.configuration.tests.excludedemo config:

```
>>> from zope.configuration.xmlconfig import string
>>> _ = string('<include package="zope.configuration.tests.excludedemo" />')
>>> len(handler.buffer)
3
>>> logged = [x.msg for x in handler.buffer]
>>> logged[0].startswith('include ')
True
>>> logged[0].endswith('zope/configuration/tests/excludedemo/configure.zcml')
True
>>> logged[1].startswith('include ')
True
>>> logged[1].endswith('zope/configuration/tests/excludedemo/sub/configure.zcml')
True
>>> logged[2].startswith('include ')
True
>>> logged[2].endswith('zope/configuration/tests/excludedemo/spam.zcml')
True
>>> del handler.buffer[:]
```

Each run of the configuration machinery runs with fresh state, so rerunning gives the same thing:

```
>>> _ = string('<include package="zope.configuration.tests.excludedemo" />')
>>> len(handler.buffer)
3
>>> logged = [x.msg for x in handler.buffer]
>>> logged[0].startswith('include ')
True
>>> logged[0].endswith('zope/configuration/tests/excludedemo/configure.zcml')
True
>>> logged[1].startswith('include ')
True
>>> logged[1].endswith('zope/configuration/tests/excludedemo/sub/configure.zcml')
True
>>> logged[2].startswith('include ')
True
>>> logged[2].endswith('zope/configuration/tests/excludedemo/spam.zcml')
True
>>> del handler.buffer[:]
```

Now, we'll use the exclude directive to exclude the two files included by the configuration file in zope.configuration.tests.excludedemo:

```
>>> _ = string(
... '''
... <configure  xmlns="http://namespaces.zope.org/zope">
...   <exclude package="zope.configuration.tests.excludedemo.sub" />
...   <exclude package="zope.configuration.tests.excludedemo" file="spam.zcml" />
...   <include package="zope.configuration.tests.excludedemo" />
... </configure>
... ''')
>>> len(handler.buffer)
1
>>> logged = [x.msg for x in handler.buffer]
>>> logged[0].startswith('include ')
True
>>> logged[0].endswith('zope/configuration/tests/excludedemo/configure.zcml')
True
```

Creating simple directives

A simple directive is a directive that doesn't contain other directives. It can be implemented via a fairly simple function. To implement a simple directive, you need to do 3 things:

- You need to create a schema to describe the directive parameters,
- You need to write a directive handler, and
- You need to register the directive.

In this example, we'll implement a contrived example that records information about files in a file registry. The file registry is just the list, `file_registry`.

```
>>> from zope.configuration.tests.simple import file_registry
```

Our registry will contain tuples with:

- file path
- file title
- description
- Information about where the file was defined

Our schema is defined in `zope.configuration.tests.simple.IRegisterFile` (q.v).

```
>>> from zope.configuration.tests.simple import IRegisterFile
```

Our schema lists the `path` and `title` attributes. We'll get the description and other information for free, as we'll see later. The title is not required, and may be omitted.

The job of a configuration handler is to compute one or more configuration actions. Configuration actions are defered function calls. The handler doesn't perform the actions. It just computes actions, which may be performed later if they are not overridden by other directives.

Our handler is given in the function, `zope.configuration.tests.simple.registerFile`.

```
>>> from zope.configuration.tests.simple import registerFile
```

It takes a context, a path and a title. All directive handlers take the directive context as the first argument. A directive context, at a minimim, implements, `zope.configuration.IConfigurationContext`. (Specialized contexts can implement more specific interfaces. We'll say more about that when we talk about grouping directives.) The title argument must have a default value, because we indicated that the title was not required in the schema. (Alternatively, we could have made the title required, but provided a default value in the schema.

In the first line of function `registerFile`, we get the context information object. This object contains information about the configuration directive, such as the file and location within the file of the directive.

The context information object also has a text attribute that contains the textual data contained by the configuration directive. (This is the concatenation of all of the xml text nodes directly contained by the directive.) We use this for our description in the second line of the handler.

The last thing the handler does is to compute an action by calling the action method of the context. It passes the action method 3 keyword arguments:

- discriminator

  The discriminator is used to identify the action to be performed so that duplicate actions can be detected. Two actions are duplicated, and this conflict, if they have the same discriminator values and the values are not `None`. Conflicting actions can be resolved if one of the conflicting actions is from a configuration file that directly or indirectly includes the files containing the other conflicting actions.

  In function `registerFile`, we a tuple with the string `'RegisterFile'` and the path to be registered.

- callable

  The callable is the object to be called to perform the action.

- args

  The args argument contains positinal arguments to be passed to the callable. In function `registerFile`, we pass a tuple containing a `FileInfo` object.

  (Note that there's nothing special about the FileInfo class. It has nothing to do with creating simple directives. It's just used in this example to organize the application data.)

The final step in implementing the simple directive is to register it. We do that with the zcml `meta:directive` directive. This is given in the file simple.zcml. Here we specify the name, namespace, schema, and handler for the directive. We also provide a documentation for the directive as text between the start and end tags.

The file simple.zcml also includes some directives that use the new directive to register some files.

Now let's try it all out:

```
>>> from zope.configuration import tests
>>> from zope.configuration.xmlconfig import file
>>> context = file("simple.zcml", tests)
```

Now we should see some file information in the registry:

```
>>> from zope.configuration.tests.test_xmlconfig import clean_text_w_paths
>>> from zope.configuration.tests.test_xmlconfig import clean_path
>>> print clean_path(file_registry[0].path)
tests/simple.py
>>> print file_registry[0].title
How to create a simple directive
>>> print file_registry[0].description
Describes how to implement a simple directive
>>> print clean_text_w_paths(file_registry[0].info)
File "tests/simple.zcml", line 19.2-24.2
    <files:register
        path="simple.py"
        title="How to create a simple directive"
        >
      Describes how to implement a simple directive
    </files:register>
>>> print clean_path(file_registry[1].path)
tests/simple.zcml
>>> print file_registry[1].title

>>> desc = file_registry[1].description
>>> print '\n'.join([l.rstrip()
...                  for l in desc.strip().splitlines()
...                    if l.rstrip()])
Shows the ZCML directives needed to register a simple directive.
    Also show some usage examples,
>>> print clean_text_w_paths(file_registry[1].info)
```

```
File "tests/simple.zcml", line 26.2-30.2
    <files:register path="simple.zcml">
      Shows the ZCML directives needed to register a simple directive.
      Also show some usage examples,
    </files:register>
>>> print clean_path(file_registry[2].path)
tests/__init__.py
>>> print file_registry[2].title
Make this a package
>>> print file_registry[2].description

>>> print clean_text_w_paths(file_registry[2].info)
File "tests/simple.zcml", line 32.2-32.67
    <files:register path="__init__.py" title="Make this a package" />
```

Clean up after ourselves:

```
>>> del file_registry[:]
```

Creating nested directives

When using ZCML, you sometimes nest ZCML directives. This is typically done either to:

- Avoid repetative input. Information shared among multiple directives is provided in a surrounding directive.
- Put together information that is too complex or structured to express with a single set of directive parameters.

Grouping directives are used to handle both of these cases. See the documentation in `zope.configure.zopeconfigure`. This file describes the implementation of the zope `configure` directive, which groups directives that use a common package or internationalization domain. You should also have read the section on "Creating simple directives."

This file shows you how to handle the second case above. In this case, we have grouping directives that are meant to collaborate with specific contained directives. To do this, you have the grouping directives declare a more specific (or alternate) interface to `IConfigurationContext`. Directives designed to work with those grouping directives are registered for the new interface.

Let's look at example. Suppose we wanted to be able to define schema using ZCML. We'd use a grouping directive to specify schemas and contained directives to specify fields within the schema. We'll use a schema registry to hold the defined schemas:

```
.. doctest::
```

```
>>> from zope.configuration.tests.nested import schema_registry
```

A schema has a name, an id, some documentation, and some fields. We'll provide the name and the id as parameters. We'll define fields as subdirectives and documentation as text contained in the schema directive. The schema directive uses the schema, `ISchemaInfo` for it's parameters.

```
>>> from zope.configuration.tests.nested import ISchemaInfo
```

We also define the schema, ISchema, that specifies an attribute that nested field directives will use to store the fields they define.

```
>>> from zope.configuration.tests.nested import ISchema
```

The class, `Schema`, provides the handler for the schema directive. (If you haven't read the documentation in `zopeconfigure.py`, you need to do so now.) The constructor saves its arguments as attributes and initializes its `fields` attribute:

```
>>> from zope.configuration.tests.nested import Schema
```

The `after` method of the `Schema` class creates a schema and computes an action to register the schema in the schema registry. The discriminator prevents two schema directives from registering the same schema.

It's important to note that when we call the `action` method on `self`, rather than on `self.context`. This is because, in a grouping directive handler, the handler instance is itself a context. When we call the `action` method, the method stores additional meta data associated with the context it was called on. This meta data includes an include path, used when resolving conflicting actions, and an object that contains information about the XML source used to invole the directive. If we called the action method on `self.context`, the wrong meta data would be associated with the configuration action.

The file `schema.zcml` contains the meta-configuration directive that defines the schema directive.

To define fields, we'll create directives to define the fields. Let's start with a `text` field. `ITextField` defines the schema for text field parameters. It extends `IFieldInfo`, which defines data common to all fields. We also define a simple handler method, textField, that takes a context and keyword arguments. (For information on writing simple directives, see `test_simple.py`.) We've abstracted most of the logic into the function `field`.

The `field` function computes a field instance using the constructor, and the keyword arguments passed to it. It also uses the context information object to get the text content of the directive, which it uses for the field description.

After computing the field instance, it gets the `Schema` instance, which is the context of the context passed to the function. The function checks to see if there is already a field with that name. If there is, it raises an error. Otherwise, it saves the field.

We also define an `IIntInfo` schema and `intField` handler function to support defining integer fields.

We register the `text` and `int` directives in `schema.zcml`. These are like the simple directive definition we saw in `test_simple.py` with an important exception. We provide a `usedIn` parameter to say that these directives can *only* ne used in a `ISchema` context. In other words, these can only be used inside of `schema` directives.

The `schema.zcml` file also contains some sample `schema` directives. We can execute the file:

```
>>> from zope.configuration import tests
>>> from zope.configuration.xmlconfig import file
>>> context = file("schema.zcml", tests)
```

And verify that the schema registery has the schemas we expect:

```
>>> pprint(sorted(schema_registry))
['zope.configuration.tests.nested.I1',
 'zope.configuration.tests.nested.I2']

>>> def sorted(x):
...     r = list(x)
...     r.sort()
...     return r

>>> i1 = schema_registry['zope.configuration.tests.nested.I1']
>>> sorted(i1)
['a', 'b']
>>> i1['a'].__class__.__name__
'Text'
>>> i1['a'].description.strip()
u'A\n\n          Blah blah'
```

```
>>> i1['a'].min_length
1
>>> i1['b'].__class__.__name__
'Int'
>>> i1['b'].description.strip()
u'B\n\n          Not feeling very creative'
>>> i1['b'].min
1
>>> i1['b'].max
10

>>> i2 = schema_registry['zope.configuration.tests.nested.I2']
>>> sorted(i2)
['x', 'y']
```

Now let's look at some error situations. For example, let's see what happens if we use a field directive outside of a schema dorective. (Note that we used the context we created above, so we don't have to redefine our directives:

```
>>> from zope.configuration.xmlconfig import string
>>> from zope.configuration.xmlconfig import ZopeXMLConfigurationError
>>> try:
...     v = string(
...       '<text xmlns="http://sample.namespaces.zope.org/schema" name="x" />',
...       context)
... except ZopeXMLConfigurationError, v:
...   pass
>>> print v
File "<string>", line 1.0
    ConfigurationError: The directive (u'http://sample.namespaces.zope.org/schema', u
↪'text') cannot be used in this context
```

Let's see what happens if we declare duplicate fields:

```
>>> try:
...     v = string(
...       '''
...       <schema name="I3" id="zope.configuration.tests.nested.I3"
...               xmlns="http://sample.namespaces.zope.org/schema">
...         <text name="x" />
...         <text name="x" />
...       </schema>
...       ''',
...       context)
... except ZopeXMLConfigurationError, v:
...   pass
>>> print v
File "<string>", line 5.7-5.24
    ValueError: ('Duplicate field', 'x')
```

CHAPTER 2

zope.configuration API Reference

CHAPTER 3

Hacking on `zope.configuration`

Getting the Code

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

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

You can get a read-only checkout from there:

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

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

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

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

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

You can branch the trunk from there using Bazaar:

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

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.configuration
```

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

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

Running the tests

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

```
$ /tmp/hack-zope.configuration/bin/python setup.py test
running test
........
----------------------------------------------------------------------
Ran 249 tests in 0.366s

OK
```

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

```
$ /tmp/hack-zope.configuration/bin/easy_install nose
...
$ /tmp/hack-zope.configuration/bin/python setup.py nosetests
running nosetests
.......
----------------------------------------------------------------------
Ran 249 tests in 0.366s

OK
```

or:

```
$ /tmp/hack-zope.configuration/bin/nosetests
.......
----------------------------------------------------------------------
Ran 249 tests in 0.366s

OK
```

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

```
$ /tmp/hack-zope.configuration/bin/easy_install nose coverage
...
$ /tmp/hack-zope.configuration/bin/python setup.py nosetests \
    --with coverage --cover-package=zope.configuration
running nosetests
...
Name                                Stmts   Miss  Cover   Missing
-----------------------------------------------------------------
zope.configuration                      3      0   100%
zope.configuration._compat              2      0   100%
zope.configuration.config             439      0   100%
zope.configuration.docutils            34      0   100%
zope.configuration.exceptions           2      0   100%
zope.configuration.fields             111      0   100%
zope.configuration.interfaces          18      0   100%
zope.configuration.name                54      0   100%
zope.configuration.xmlconfig          269      0   100%
zope.configuration.zopeconfigure       17      0   100%
-----------------------------------------------------------------
```

```
TOTAL                                   955      0   100%
----------------------------------------------------------------------
Ran 256 tests in 1.063s

OK
```

Building the documentation

`zope.configuration` 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.configuration/bin/easy_install Sphinx
...
$ cd docs
$ PATH=/tmp/hack-zope.configuration/bin:$PATH make html
sphinx-build -b html -d _build/doctrees   . _build/html
...
build succeeded.

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

You can also test the code snippets in the documentation:

```
$ PATH=/tmp/hack-zope.configuration/bin:$PATH make doctest
sphinx-build -b doctest -d _build/doctrees   . _build/doctest
...

Doctest summary
===============
  554 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.configuration` 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/configuration/.'
...
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 249 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.configuration` 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.configuration` and dependencies, and runs the tests via `python setup.py test -q`.
- The `coverage` environment builds a `virtualenv` with `python2.6`, installs `zope.configuration`, installs `nose` and `coverage`, and runs `nosetests` with statement coverage.
- The `docs` environment builds a virtualenv with `python2.6`, installs `zope.configuration`, 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 249 tests in 0.366s

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
===============
 544 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.configuration`

Submitting a Bug Report

`zope.configuration` tracks its bugs on Github:

> https://github.com/zopefoundation/zope.configuration/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.configuration/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.configuration/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

Python Module Index

z

Index

Z