zope.configuration Documentation

Release 4.0

Zope Foundation Contributors

January 28, 2015

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.

1.1 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}]
```

1.2 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[:]
```

1.3 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
```

1.4 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
```

1.5 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[:]
```

1.6 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 h
```

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

2.1 zope.configuration.config

class zope.configuration.config.**ConfigurationContext**

Mix-in that implements IConfigurationContext

Subclasses provide a `package` attribute and a `basepath` attribute. If the base path is not None, relative paths are converted to absolute paths using the the base path. If the package is not none, relative imports are performed relative to the package.

In general, the basepath and package attributes should be consistent. When a package is provided, the base path should be set to the path of the package directory.

Subclasses also provide an `actions` attribute, which is a list of actions, an `includepath` attribute, and an `info` attribute.

The include path is appended to each action and is used when resolving conflicts among actions. Normally, only the a ConfigurationMachine provides the actions attribute. Decorators simply use the actions of the context they decorate. The `includepath` attribute is a tuple of names. Each name is typically the name of an included configuration file.

The `info` attribute contains descriptive information helpful when reporting errors. If not set, it defaults to an empty string.

The actions attribute is a sequence of dictionaries where each dictionary has the following keys:

- `discriminator`, a value that identifies the action. Two actions that have the same (non None) discriminator conflict.
- `callable`, an object that is called to execute the action,
- `args`, positional arguments for the action
- `kw`, keyword arguments for the action
- `includepath`, a tuple of include file names (defaults to ())
- `info`, an object that has descriptive information about the action (defaults to '')

resolve(*dottedname*)

Resolve a dotted name to an object.

Examples:

```
>>> from zope.configuration.config import ConfigurationContext
>>> from zope.configuration.config import ConfigurationError
>>> c = ConfigurationContext()
```

```
>>> import zope, zope.interface
>>> c.resolve('zope') is zope
True
>>> c.resolve('zope.interface') is zope.interface
True
>>> c.resolve('zope.configuration.eek')
Traceback (most recent call last):
...
ConfigurationError:
ImportError: Module zope.configuration has no global eek

>>> c.resolve('.config.ConfigurationContext')
Traceback (most recent call last):
...
AttributeError: 'ConfigurationContext' object has no attribute 'package'
>>> import zope.configuration
>>> c.package = zope.configuration
>>> c.resolve('.') is zope.configuration
True
>>> c.resolve('.config.ConfigurationContext') is ConfigurationContext
True
>>> c.resolve('..interface') is zope.interface
True
>>> c.resolve('str')
<type 'str'>
```

path (*filename*)

Compute package-relative paths.

Examples:

```
>>> import os
>>> from zope.configuration.config import ConfigurationContext
>>> c = ConfigurationContext()
>>> c.path("/x/y/z") == os.path.normpath("/x/y/z")
True
>>> c.path("y/z")
Traceback (most recent call last):
...
AttributeError: 'ConfigurationContext' object has no attribute 'package'
>>> import zope.configuration
>>> c.package = zope.configuration
>>> import os
>>> d = os.path.dirname(zope.configuration.__file__)
>>> c.path("y/z") == d + os.path.normpath("/y/z")
True
>>> c.path("y/./z") == d + os.path.normpath("/y/z")
True
>>> c.path("y/../z") == d + os.path.normpath("/z")
True
```

checkDuplicate (*filename*)

Check for duplicate imports of the same file.

Raises an exception if this file had been processed before. This is better than an unlimited number of conflict errors.

Examples:

```
>>> from zope.configuration.config import ConfigurationContext
>>> from zope.configuration.config import ConfigurationError
>>> c = ConfigurationContext()
>>> c.checkDuplicate('/foo.zcml')
>>> try:
...     c.checkDuplicate('/foo.zcml')
... except ConfigurationError as e:
...     # On Linux the exact msg has /foo, on Windows \foo.
...     str(e).endswith("foo.zcml' included more than once")
True
```

You may use different ways to refer to the same file:

```
>>> import zope.configuration
>>> c.package = zope.configuration
>>> import os
>>> d = os.path.dirname(zope.configuration.__file__)
>>> c.checkDuplicate('bar.zcml')
>>> try:
...   c.checkDuplicate(d + os.path.normpath('/bar.zcml'))
... except ConfigurationError as e:
...   str(e).endswith("bar.zcml' included more than once")
...
True
```

processFile(*filename*)

Check whether a file needs to be processed.

Return True if processing is needed and False otherwise. If the file needs to be processed, it will be marked as processed, assuming that the caller will procces the file if it needs to be procssed.

Examples:

```
>>> from zope.configuration.config import ConfigurationContext
>>> c = ConfigurationContext()
>>> c.processFile('/foo.zcml')
True
>>> c.processFile('/foo.zcml')
False
```

You may use different ways to refer to the same file:

```
>>> import zope.configuration
>>> c.package = zope.configuration
>>> import os
>>> d = os.path.dirname(zope.configuration.__file__)
>>> c.processFile('bar.zcml')
True
>>> c.processFile('bar.zcml')
False
```

action(*discriminator*, *callable=None*, *args=()*, *kw=None*, *order=0*, *includepath=None*, *info=None*, ***extra*)

Add an action with the given discriminator, callable and arguments.

For testing purposes, the callable and arguments may be omitted. In that case, a default noop callable is used.

The discriminator must be given, but it can be None, to indicate that the action never conflicts.

Examples:

```
>>> from zope.configuration.config import ConfigurationContext
>>> c = ConfigurationContext()
```

Normally, the context gets actions from subclasses. We'll provide an actions attribute ourselves:

```
>>> c.actions = []
```

We'll use a test callable that has a convenient string representation

```
>>> from zope.configuration.tests.directives import f
>>> c.action(1, f, (1, ), {'x': 1})
>>> from pprint import PrettyPrinter
>>> pprint=PrettyPrinter(width=60).pprint
>>> pprint(c.actions)
[{'args': (1,),
  'callable': f,
  'discriminator': 1,
  'includepath': (),
  'info': '',
  'kw': {'x': 1},
  'order': 0}]

>>> c.action(None)
>>> pprint(c.actions)
[{'args': (1,),
  'callable': f,
  'discriminator': 1,
  'includepath': (),
  'info': '',
  'kw': {'x': 1},
  'order': 0},
 {'args': (),
  'callable': None,
  'discriminator': None,
  'includepath': (),
  'info': '',
  'kw': {},
  'order': 0}]
```

Now set the include path and info:

```
>>> c.includepath = ('foo.zcml',)
>>> c.info = "?"
>>> c.action(None)
>>> pprint(c.actions[-1])
{'args': (),
 'callable': None,
 'discriminator': None,
 'includepath': ('foo.zcml',),
 'info': '?',
 'kw': {},
 'order': 0}
```

We can add an order argument to crudely control the order of execution:

```
>>> c.action(None, order=99999)
>>> pprint(c.actions[-1])
{'args': (),
 'callable': None,
 'discriminator': None,
```

```
 'includepath': ('foo.zcml',),
 'info': '?',
 'kw': {},
 'order': 99999}
```

We can also pass an includepath argument, which will be used as the the includepath for the action. (if includepath is None, self.includepath will be used):

```
>>> c.action(None, includepath=('abc',))
>>> pprint(c.actions[-1])
{'args': (),
 'callable': None,
 'discriminator': None,
 'includepath': ('abc',),
 'info': '?',
 'kw': {},
 'order': 0}
```

We can also pass an info argument, which will be used as the the source line info for the action. (if info is None, self.info will be used):

```
>>> c.action(None, info='abc')
>>> pprint(c.actions[-1])
{'args': (),
 'callable': None,
 'discriminator': None,
 'includepath': ('foo.zcml',),
 'info': 'abc',
 'kw': {},
 'order': 0}
```

hasFeature (*feature*)

Check whether a named feature has been provided.

Initially no features are provided

Examples:

```
>>> from zope.configuration.config import ConfigurationContext
>>> c = ConfigurationContext()
>>> c.hasFeature('onlinehelp')
False
```

You can declare that a feature is provided

```
>>> c.provideFeature('onlinehelp')
```

and it becomes available

```
>>> c.hasFeature('onlinehelp')
True
```

provideFeature (*feature*)

Declare thata named feature has been provided.

See `hasFeature()` for examples.

class zope.configuration.config.**ConfigurationAdapterRegistry**

Simple adapter registry that manages directives as adapters

Examples:

```
>>> from zope.configuration.interfaces import IConfigurationContext
>>> from zope.configuration.config import ConfigurationAdapterRegistry
>>> from zope.configuration.config import ConfigurationError
>>> from zope.configuration.config import ConfigurationMachine
>>> r = ConfigurationAdapterRegistry()
>>> c = ConfigurationMachine()
>>> r.factory(c, ('http://www.zope.com','xxx'))
Traceback (most recent call last):
...
ConfigurationError: ('Unknown directive', 'http://www.zope.com', 'xxx')
>>> def f():
...     pass

>>> r.register(IConfigurationContext, ('http://www.zope.com', 'xxx'), f)
>>> r.factory(c, ('http://www.zope.com','xxx')) is f
True
>>> r.factory(c, ('http://www.zope.com','yyy')) is f
Traceback (most recent call last):
...
ConfigurationError: ('Unknown directive', 'http://www.zope.com', 'yyy')
>>> r.register(IConfigurationContext, 'yyy', f)
>>> r.factory(c, ('http://www.zope.com','yyy')) is f
True
```

Test the documentation feature:

```
>>> from zope.configuration.config import IFullInfo
>>> r._docRegistry
[]
>>> r.document(('ns', 'dir'), IFullInfo, IConfigurationContext, None,
...            'inf', None)
>>> r._docRegistry[0][0] == ('ns', 'dir')
True
>>> r._docRegistry[0][1] is IFullInfo
True
>>> r._docRegistry[0][2] is IConfigurationContext
True
>>> r._docRegistry[0][3] is None
True
>>> r._docRegistry[0][4] == 'inf'
True
>>> r._docRegistry[0][5] is None
True
>>> r.document('all-dir', None, None, None, None)
>>> r._docRegistry[1][0]
('', 'all-dir')
```

class zope.configuration.config.**ConfigurationMachine**

Configuration machine

Example:

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

Register a directive:

```
>>> from zope.configuration.config import metans
>>> machine((metans, "directive"),
...         namespace=ns, name="simple",
...         schema="zope.configuration.tests.directives.ISimple",
...         handler="zope.configuration.tests.directives.simple")
```

and try it out:

```
>>> machine((ns, "simple"), a=u"aa", c=u"cc")
>>> from pprint import PrettyPrinter
>>> pprint = PrettyPrinter(width=60).pprint
>>> pprint(machine.actions)
[{'args': (u'aa', u'xxx', 'cc'),
  'callable': f,
  'discriminator': ('simple', u'aa', u'xxx', 'cc'),
  'includepath': (),
  'info': None,
  'kw': {},
  'order': 0}]
```

begin(*_ConfigurationMachine__name*, *_ConfigurationMachine__data=None*, *_ConfigurationMachine__info=None*, ***kw*)

end()

__call__(*_ConfigurationMachine__name*, *_ConfigurationMachine__info=None*, ***_ConfigurationMachine__kw*)

getInfo()

setInfo(*info*)

execute_actions(*clear=True*, *testing=False*)

Execute the configuration actions.

This calls the action callables after resolving conflicts.

For example:

```
>>> from zope.configuration.config import ConfigurationMachine
>>> output = []
>>> def f(*a, **k): #* syntax highlighting
...     output.append(('f', a, k))
>>> context = ConfigurationMachine()
>>> context.actions = [
...    (1, f, (1,)),
...    (1, f, (11,), {}, ('x', )),
...    (2, f, (2,)),
...    ]
>>> context.execute_actions()
>>> output
[('f', (1,), {}), ('f', (2,), {})]
```

If the action raises an error, we convert it to a ConfigurationExecutionError.

```
>>> from zope.configuration.config import ConfigurationExecutionError
>>> output = []
>>> def bad():
...     bad.xxx
>>> context.actions = [
...    (1, f, (1,)),
...    (1, f, (11,), {}, ('x', )),
```

```
...    (2, f, (2,)),
...    (3, bad, (), {}, (), 'oops')
...    ]
>>> try:
...     v = context.execute_actions()
... except ConfigurationExecutionError as v:
...     pass
>>> lines = str(v).splitlines()
>>> 'exceptions.AttributeError' in lines[0]
True
>>> lines[0].endswith("'function' object has no attribute 'xxx'")
True
>>> lines[1:]
['  in:', '  oops']
```

Note that actions executed before the error still have an effect:

```
>>> output
[('f', (1,), {}), ('f', (2,), {})]
```

class zope.configuration.config.**ConfigurationExecutionError**(*etype*, *evalue*, *info*)
An error occurred during execution of a configuration action

interface zope.configuration.config.**IStackItem**
Configuration machine stack items

Stack items are created when a directive is being processed.

A stack item is created for each directive use.

contained(*name*, *data*, *info*)
Begin processing a contained directive

The data are a dictionary of attribute names mapped to unicode strings.

The info argument is an object that can be converted to a string and that contains information about the directive.

The begin method returns the next item to be placed on the stack.

finish()
Finish processing a directive

class zope.configuration.config.**SimpleStackItem**(*context*, *handler*, *info*, **argdata*)
Simple stack item

A simple stack item can't have anything added after it. It can only be removed. It is used for simple directives and subdirectives, which can't contain other directives.

It also defers any computation until the end of the directive has been reached.

class zope.configuration.config.**RootStackItem**(*context*)

contained(*name*, *data*, *info*)
Handle a contained directive

We have to compute a new stack item by getting a named adapter for the current context object.

class zope.configuration.config.**GroupingStackItem**(*context*)
Stack item for a grouping directive

A grouping stack item is in the stack when a grouping directive is being processed. Grouping directives group other directives. Often, they just manage common data, but they may also take actions, either before or after contained directives are executed.

A grouping stack item is created with a grouping directive definition, a configuration context, and directive data.

To see how this works, let's look at an example:

We need a context. We'll just use a configuration machine

```
>>> from zope.configuration.config import GroupingStackItem
>>> from zope.configuration.config import ConfigurationMachine
>>> context = ConfigurationMachine()
```

We need a callable to use in configuration actions. We'll use a convenient one from the tests:

```
>>> from zope.configuration.tests.directives import f
```

We need a handler for the grouping directive. This is a class that implements a context decorator. The decorator must also provide `before` and `after` methods that are called before and after any contained directives are processed. We'll typically subclass `GroupingContextDecorator`, which provides context decoration, and default `before` and `after` methods.

```
>>> from zope.configuration.config import GroupingContextDecorator
>>> class SampleGrouping(GroupingContextDecorator):
...     def before(self):
...         self.action(('before', self.x, self.y), f)
...     def after(self):
...         self.action(('after'), f)
```

We'll use our decorator to decorate our initial context, providing keyword arguments x and y:

```
>>> dec = SampleGrouping(context, x=1, y=2)
```

Note that the keyword arguments are made attributes of the decorator.

Now we'll create the stack item.

```
>>> item = GroupingStackItem(dec)
```

We still haven't called the before action yet, which we can verify by looking at the context actions:

```
>>> context.actions
[]
```

Subdirectives will get looked up as adapters of the context.

We'll create a simple handler:

```
>>> def simple(context, data, info):
...     context.action(("simple", context.x, context.y, data), f)
...     return info
```

and register it with the context:

```
>>> from zope.configuration.interfaces import IConfigurationContext
>>> from zope.configuration.config import testns
>>> context.register(IConfigurationContext, (testns, 'simple'), simple)
```

This handler isn't really a propert handler, because it doesn't return a new context. It will do for this example.

Now we'll call the contained method on the stack item:

```
>>> item.contained((testns, 'simple'), {'z': 'zope'}, "someinfo")
'someinfo'
```

We can verify thet the simple method was called by looking at the context actions. Note that the before method was called before handling the contained directive.

```
>>> from pprint import PrettyPrinter
>>> pprint = PrettyPrinter(width=60).pprint

>>> pprint(context.actions)
[{'args': (),
  'callable': f,
  'discriminator': ('before', 1, 2),
  'includepath': (),
  'info': '',
  'kw': {},
  'order': 0},
 {'args': (),
  'callable': f,
  'discriminator': ('simple', 1, 2, {'z': 'zope'}),
  'includepath': (),
  'info': '',
  'kw': {},
  'order': 0}]
```

Finally, we call finish, which calls the decorator after method:

```
>>> item.finish()

>>> pprint(context.actions)
[{'args': (),
  'callable': f,
  'discriminator': ('before', 1, 2),
  'includepath': (),
  'info': '',
  'kw': {},
  'order': 0},
 {'args': (),
  'callable': f,
  'discriminator': ('simple', 1, 2, {'z': 'zope'}),
  'includepath': (),
  'info': '',
  'kw': {},
  'order': 0},
 {'args': (),
  'callable': f,
  'discriminator': 'after',
  'includepath': (),
  'info': '',
  'kw': {},
  'order': 0}]
```

If there were no nested directives:

```
>>> context = ConfigurationMachine()
>>> dec = SampleGrouping(context, x=1, y=2)
>>> item = GroupingStackItem(dec)
>>> item.finish()
```

Then before will be when we call finish:

```
>>> pprint(context.actions)
[{'args': (),
  'callable': f,
  'discriminator': ('before', 1, 2),
  'includepath': (),
  'info': '',
  'kw': {},
  'order': 0},
 {'args': (),
  'callable': f,
  'discriminator': 'after',
  'includepath': (),
  'info': '',
  'kw': {},
  'order': 0}]
```

class zope.configuration.config.**ComplexStackItem**(*meta*, *context*, *data*, *info*)

Complex stack item

A complex stack item is in the stack when a complex directive is being processed. It only allows subdirectives to be used.

A complex stack item is created with a complex directive definition (IComplexDirectiveContext), a configuration context, and directive data.

To see how this works, let's look at an example:

We need a context. We'll just use a configuration machine

```
>>> from zope.configuration.config import ConfigurationMachine
>>> context = ConfigurationMachine()
```

We need a callable to use in configuration actions. We'll use a convenient one from the tests:

```
>>> from zope.configuration.tests.directives import f
```

We need a handler for the complex directive. This is a class with a method for each subdirective:

```
>>> class Handler(object):
...   def __init__(self, context, x, y):
...      self.context, self.x, self.y = context, x, y
...      context.action('init', f)
...   def sub(self, context, a, b):
...      context.action(('sub', a, b), f)
...   def __call__(self):
...      self.context.action(('call', self.x, self.y), f)
```

We need a complex directive definition:

```
>>> from zope.interface import Interface
>>> from zope.schema import TextLine
>>> from zope.configuration.config import ComplexDirectiveDefinition
>>> class Ixy(Interface):
...    x = TextLine()
...    y = TextLine()
>>> definition = ComplexDirectiveDefinition(
...        context, name="test", schema=Ixy,
...        handler=Handler)
>>> class Iab(Interface):
...    a = TextLine()
```

```
...     b = TextLine()
>>> definition['sub'] = Iab, ''
```

OK, now that we have the context, handler and definition, we're ready to use a stack item.

```
>>> from zope.configuration.config import ComplexStackItem
>>> item = ComplexStackItem(definition, context, {'x': u'xv', 'y': u'yv'},
...                         'foo')
```

When we created the definition, the handler (factory) was called.

```
>>> from pprint import PrettyPrinter
>>> pprint = PrettyPrinter(width=60).pprint
>>> pprint(context.actions)
[{'args': (),
  'callable': f,
  'discriminator': 'init',
  'includepath': (),
  'info': 'foo',
  'kw': {},
  'order': 0}]
```

If a subdirective is provided, the `contained` method of the stack item is called. It will lookup the subdirective schema and call the corresponding method on the handler instance:

```
>>> simple = item.contained(('somenamespace', 'sub'),
...                         {'a': u'av', 'b': u'bv'}, 'baz')
>>> simple.finish()
```

Note that the name passed to `contained` is a 2-part name, consisting of a namespace and a name within the namespace.

```
>>> pprint(context.actions)
[{'args': (),
  'callable': f,
  'discriminator': 'init',
  'includepath': (),
  'info': 'foo',
  'kw': {},
  'order': 0},
 {'args': (),
  'callable': f,
  'discriminator': ('sub', u'av', u'bv'),
  'includepath': (),
  'info': 'baz',
  'kw': {},
  'order': 0}]
```

The new stack item returned by contained is one that doesn't allow any more subdirectives,

When all of the subdirectives have been provided, the `finish` method is called:

```
>>> item.finish()
```

The stack item will call the handler if it is callable.

```
>>> pprint(context.actions)
[{'args': (),
  'callable': f,
  'discriminator': 'init',
```

```
  'includepath': (),
  'info': 'foo',
  'kw': {},
  'order': 0},
 {'args': (),
  'callable': f,
  'discriminator': ('sub', u'av', u'bv'),
  'includepath': (),
  'info': 'baz',
  'kw': {},
  'order': 0},
 {'args': (),
  'callable': f,
  'discriminator': ('call', u'xv', u'yv'),
  'includepath': (),
  'info': 'foo',
  'kw': {},
  'order': 0}]
```

contained(*name*, *data*, *info*)
Handle a subdirective

class zope.configuration.config.**GroupingContextDecorator**(*context*, ***kw*)
Helper mix-in class for building grouping directives

See the discussion (and test) in GroupingStackItem.

class zope.configuration.config.**DirectiveSchema**(***kw*)
A field that contains a global variable value that must be a schema

interface zope.configuration.config.**IDirectivesInfo**
Schema for the directives directive

namespace
Namespace

The namespace in which directives' names will be defined

interface zope.configuration.config.**IDirectivesContext**
Extends: zope.configuration.config.IDirectivesInfo, zope.configuration.interfaces.IConfigu

class zope.configuration.config.**DirectivesHandler**(*context*, ***kw*)
Handler for the directives directive

This is just a grouping directive that adds a namespace attribute to the normal directive context.

interface zope.configuration.config.**IDirectiveInfo**
Information common to all directive definitions have

name
Directive name

The name of the directive being defined

schema
Directive handler

The dotted name of the directive handler

interface zope.configuration.config.**IFullInfo**
Extends: zope.configuration.config.IDirectiveInfo

Information that all top-level directives (not subdirectives) have

handler
Directive handler

The dotted name of the directive handler

usedIn
The directive types the directive can be used in

The interface of the directives that can contain the directive

interface zope.configuration.config.**IStandaloneDirectiveInfo**
Extends: zope.configuration.config.IDirectivesInfo, zope.configuration.config.IFullInfo

Info for full directives defined outside a directives directives

zope.configuration.config.**defineSimpleDirective**(*context*, *name*, *schema*, *handler*, *namespace=''*, *usedIn=<InterfaceClass zope.configuration.interfaces.IConfigurationContext>*)

Define a simple directive

Define and register a factory that invokes the simple directive and returns a new stack item, which is always the same simple stack item.

If the namespace is '*', the directive is registered for all namespaces.

Example:

```
>>> from zope.configuration.config import ConfigurationMachine
>>> context = ConfigurationMachine()
>>> from zope.interface import Interface
>>> from zope.schema import TextLine
>>> from zope.configuration.tests.directives import f
>>> class Ixy(Interface):
...     x = TextLine()
...     y = TextLine()
>>> def s(context, x, y):
...     context.action(('s', x, y), f)

>>> from zope.configuration.config import defineSimpleDirective
>>> defineSimpleDirective(context, 's', Ixy, s, testns)

>>> context((testns, "s"), x=u"vx", y=u"vy")
>>> from pprint import PrettyPrinter
>>> pprint = PrettyPrinter(width=60).pprint
>>> pprint(context.actions)
[{'args': (),
  'callable': f,
  'discriminator': ('s', u'vx', u'vy'),
  'includepath': (),
  'info': None,
  'kw': {},
  'order': 0}]

>>> context(('http://www.zope.com/t1', "s"), x=u"vx", y=u"vy")
Traceback (most recent call last):
...
ConfigurationError: ('Unknown directive', 'http://www.zope.com/t1', 's')

>>> context = ConfigurationMachine()
>>> defineSimpleDirective(context, 's', Ixy, s, "*")
```

```
>>> context(('http://www.zope.com/t1', "s"), x=u"vx", y=u"vy")
>>> pprint(context.actions)
[{'args': (),
  'callable': f,
  'discriminator': ('s', u'vx', u'vy'),
  'includepath': (),
  'info': None,
  'kw': {},
  'order': 0}]
```

zope.configuration.config.**defineGroupingDirective**(*context*, *name*, *schema*, *handler*, *namespace=''*, *usedIn=<InterfaceClass zope.configuration.interfaces.IConfigurationContext>*)

Define a grouping directive

Define and register a factory that sets up a grouping directive.

If the namespace is '*', the directive is registered for all namespaces.

Example:

```
>>> from zope.configuration.config import ConfigurationMachine
>>> context = ConfigurationMachine()
>>> from zope.interface import Interface
>>> from zope.schema import TextLine
>>> from zope.configuration.tests.directives import f
>>> class Ixy(Interface):
...     x = TextLine()
...     y = TextLine()
```

We won't bother creating a special grouping directive class. We'll just use `GroupingContextDecorator`, which simply sets up a grouping context that has extra attributes defined by a schema:

```
>>> from zope.configuration.config import defineGroupingDirective
>>> from zope.configuration.config import GroupingContextDecorator
>>> defineGroupingDirective(context, 'g', Ixy,
...                         GroupingContextDecorator, testns)

>>> context.begin((testns, "g"), x=u"vx", y=u"vy")
>>> context.stack[-1].context.x
u'vx'
>>> context.stack[-1].context.y
u'vy'

>>> context(('http://www.zope.com/t1', "g"), x=u"vx", y=u"vy")
Traceback (most recent call last):
...
ConfigurationError: ('Unknown directive', 'http://www.zope.com/t1', 'g')

>>> context = ConfigurationMachine()
>>> defineGroupingDirective(context, 'g', Ixy,
...                         GroupingContextDecorator, "*")

>>> context.begin(('http://www.zope.com/t1', "g"), x=u"vx", y=u"vy")
>>> context.stack[-1].context.x
u'vx'
>>> context.stack[-1].context.y
u'vy'
```

interface zope.configuration.config.**IComplexDirectiveContext**
Extends: zope.configuration.config.IFullInfo, zope.configuration.interfaces.IConfiguration

class zope.configuration.config.**ComplexDirectiveDefinition**(*context*, ***kw*)
Handler for defining complex directives

See the description and tests for ComplexStackItem.

zope.configuration.config.**subdirective**(*context*, *name*, *schema*)

interface zope.configuration.config.**IProvidesDirectiveInfo**
Information for a <meta:provides> directive

feature
Feature name

The name of the feature being provided

You can test available features with zcml:condition="have featurename".

zope.configuration.config.**provides**(*context*, *feature*)
Declare that a feature is provided in context.

Example:

```
>>> from zope.configuration.config import ConfigurationContext
>>> from zope.configuration.config import provides
>>> c = ConfigurationContext()
>>> provides(c, 'apidoc')
>>> c.hasFeature('apidoc')
True
```

Spaces are not allowed in feature names (this is reserved for providing many features with a single directive in the fututе).

```
>>> provides(c, 'apidoc onlinehelp')
Traceback (most recent call last):
  ...
ValueError: Only one feature name allowed

>>> c.hasFeature('apidoc onlinehelp')
False
```

zope.configuration.config.**toargs**(*context*, *schema*, *data*)
Marshal data to an argument dictionary using a schema

Names that are python keywords have an underscore added as a suffix in the schema and in the argument list, but are used without the underscore in the data.

The fields in the schema must all implement IFromUnicode.

All of the items in the data must have corresponding fields in the schema unless the schema has a true tagged value named 'keyword_arguments'.

Example:

```
>>> from zope.configuration.config import toargs
>>> from zope.schema import BytesLine
>>> from zope.schema import Float
>>> from zope.schema import Int
>>> from zope.schema import TextLine
>>> from zope.schema import URI
>>> class schema(Interface):
```

```
...      in_ = Int(constraint=lambda v: v > 0)
...      f = Float()
...      n = TextLine(min_length=1, default=u"rob")
...      x = BytesLine(required=False)
...      u = URI()

>>> context = ConfigurationMachine()
>>> from pprint import PrettyPrinter
>>> pprint=PrettyPrinter(width=50).pprint

>>> pprint(toargs(context, schema,
...         {'in': u'1', 'f': u'1.2', 'n': u'bob', 'x': u'x.y.z',
...           'u': u'http://www.zope.org' }))
{'f': 1.2,
 'in_': 1,
 'n': u'bob',
 'u': 'http://www.zope.org',
 'x': 'x.y.z'}
```

If we have extra data, we'll get an error:

```
>>> toargs(context, schema,
...         {'in': u'1', 'f': u'1.2', 'n': u'bob', 'x': u'x.y.z',
...           'u': u'http://www.zope.org', 'a': u'1'})
Traceback (most recent call last):
...
ConfigurationError: ('Unrecognized parameters:', 'a')
```

Unless we set a tagged value to say that extra arguments are ok:

```
>>> schema.setTaggedValue('keyword_arguments', True)

>>> pprint(toargs(context, schema,
...         {'in': u'1', 'f': u'1.2', 'n': u'bob', 'x': u'x.y.z',
...           'u': u'http://www.zope.org', 'a': u'1'}))
{'a': u'1',
 'f': 1.2,
 'in_': 1,
 'n': u'bob',
 'u': 'http://www.zope.org',
 'x': 'x.y.z'}
```

If we omit required data we get an error telling us what was omitted:

```
>>> pprint(toargs(context, schema,
...         {'in': u'1', 'f': u'1.2', 'n': u'bob', 'x': u'x.y.z'}))
Traceback (most recent call last):
...
ConfigurationError: ('Missing parameter:', 'u')
```

Although we can omit not-required data:

```
>>> pprint(toargs(context, schema,
...         {'in': u'1', 'f': u'1.2', 'n': u'bob',
...           'u': u'http://www.zope.org', 'a': u'1'}))
{'a': u'1',
 'f': 1.2,
 'in_': 1,
 'n': u'bob',
 'u': 'http://www.zope.org'}
```

And we can omit required fields if they have valid defaults (defaults that are valid values):

```
>>> pprint(toargs(context, schema,
...         {'in': u'1', 'f': u'1.2',
...           'u': u'http://www.zope.org', 'a': u'1'}))
{'a': u'1',
 'f': 1.2,
 'in_': 1,
 'n': u'rob',
 'u': 'http://www.zope.org'}
```

We also get an error if any data was invalid:

```
>>> pprint(toargs(context, schema,
...         {'in': u'0', 'f': u'1.2', 'n': u'bob', 'x': u'x.y.z',
...           'u': u'http://www.zope.org', 'a': u'1'}))
Traceback (most recent call last):
...
ConfigurationError: ('Invalid value for', 'in', '0')
```

zope.configuration.config.**expand_action**(*discriminator*, *callable=None*, *args=()*, *kw=None*, *includepath=()*, *info=None*, *order=0*, ***extra*)

zope.configuration.config.**resolveConflicts**(*actions*)

Resolve conflicting actions

Given an actions list, identify and try to resolve conflicting actions. Actions conflict if they have the same non-None discriminator. Conflicting actions can be resolved if the include path of one of the actions is a prefix of the includepaths of the other conflicting actions and is unequal to the include paths in the other conflicting actions.

class zope.configuration.config.**ConfigurationConflictError**(*conflicts*)

2.2 zope.configuration.docutils

zope.configuration.docutils.**wrap**(*text*, *width=78*, *indent=0*)

Makes sure that we keep a line length of a certain width.

Examples:

```
>>> from zope.configuration.docutils import wrap
>>> print wrap('foo bar')[:-2]
foo bar
>>> print wrap('foo bar', indent=2)[:-2]
  foo bar
>>> print wrap('foo bar, more foo bar', 10)[:-2]
foo bar,
more foo
bar
>>> print wrap('foo bar, more foo bar', 10, 2)[:-2]
  foo bar,
  more foo
  bar
```

zope.configuration.docutils.**makeDocStructures**(*context*)

Creates two structures that provide a friendly format for documentation.

'namespaces' is a dictionary that maps namespaces to a directives dictionary with the key being the name of the directive and the value is a tuple: (schema, handler, info).

'subdirs' maps a (namespace, name) pair to a list of subdirectives that have the form (namespace, name, schema, info).

2.3 zope.configuration.exceptions

class zope.configuration.exceptions.**ConfigurationError**
There was an error in a configuration

2.4 zope.configuration.fields

class zope.configuration.fields.**PythonIdentifier**(**args*, ***kw*)
This field describes a python identifier, i.e. a variable name.

Let's look at an example:

```
>>> from zope.configuration.fields import PythonIdentifier
>>> class FauxContext(object):
...     pass
>>> context = FauxContext()
>>> field = PythonIdentifier().bind(context)
```

Let's test the fromUnicode method:

```
>>> field.fromUnicode(u'foo')
u'foo'
>>> field.fromUnicode(u'foo3')
u'foo3'
>>> field.fromUnicode(u'_foo3')
u'_foo3'
```

Now let's see whether validation works alright

```
>>> for value in (u'foo', u'foo3', u'foo_', u'_foo3', u'foo_3', u'foo3_'):
...     field._validate(value)
>>> from zope.schema import ValidationError
>>> for value in (u'3foo', u'foo:', u'\\', u''):
...     try:
...         field._validate(value)
...     except ValidationError:
...         print 'Validation Error'
Validation Error
Validation Error
Validation Error
Validation Error
```

class zope.configuration.fields.**GlobalObject**(*value_type=None*, ***kw*)
An object that can be accessed as a module global.

Let's look at an example:

```
>>> d = {'x': 1, 'y': 42, 'z': 'zope'}
>>> class fakeresolver(dict):
...     def resolve(self, n):
```

```
...         return self[n]
>>> fake = fakeresolver(d)

>>> from zope.schema import Int
>>> from zope.configuration.fields import GlobalObject
>>> g = GlobalObject(value_type=Int())
>>> gg = g.bind(fake)
>>> gg.fromUnicode("x")
1
>>> gg.fromUnicode("   x  \n  ")
1
>>> gg.fromUnicode("y")
42
>>> gg.fromUnicode("z")
Traceback (most recent call last):
...
WrongType: ('zope', (<type 'int'>, <type 'long'>), '')

>>> g = GlobalObject(constraint=lambda x: x%2 == 0)
>>> gg = g.bind(fake)
>>> gg.fromUnicode("x")
Traceback (most recent call last):
...
ConstraintNotSatisfied: 1
>>> gg.fromUnicode("y")
42
>>> g = GlobalObject()
>>> gg = g.bind(fake)
>>> print gg.fromUnicode('*')
None
```

class zope.configuration.fields.**GlobalInterface**(***kw*)

An interface that can be accessed from a module.

Example:

First, we need to set up a stub name resolver:

```
>>> from zope.interface import Interface
>>> class IFoo(Interface):
...     pass
>>> class Foo(object):
...     pass
>>> d = {'Foo': Foo, 'IFoo': IFoo}
>>> class fakeresolver(dict):
...     def resolve(self, n):
...         return self[n]
>>> fake = fakeresolver(d)
```

Now verify constraints are checked correctly:

```
>>> from zope.configuration.fields import GlobalInterface
>>> g = GlobalInterface()
>>> gg = g.bind(fake)
>>> gg.fromUnicode('IFoo') is IFoo
True
>>> gg.fromUnicode('  IFoo  ') is IFoo
True
>>> gg.fromUnicode('Foo')
Traceback (most recent call last):
```

```
...
WrongType: ('An interface is required', ...
```

class zope.configuration.fields.**Tokens**(*value_type=None*, *unique=False*, ***kw*)
A list that can be read from a space-separated string.

Consider GlobalObject tokens:

First, we need to set up a stub name resolver:

```
>>> d = {'x': 1, 'y': 42, 'z': 'zope', 'x.y.x': 'foo'}
>>> class fakeresolver(dict):
...     def resolve(self, n):
...         return self[n]
>>> fake = fakeresolver(d)

>>> from zope.configuration.fields import Tokens
>>> from zope.configuration.fields import GlobalObject
>>> g = Tokens(value_type=GlobalObject())
>>> gg = g.bind(fake)
>>> gg.fromUnicode("  \n  x y z  \n")
[1, 42, 'zope']

>>> from zope.schema import Int
>>> g = Tokens(value_type=
...            GlobalObject(value_type=
...                         Int(constraint=lambda x: x%2 == 0)))
>>> gg = g.bind(fake)
>>> gg.fromUnicode("x y")
Traceback (most recent call last):
...
InvalidToken: 1 in x y

>>> gg.fromUnicode("z y")
Traceback (most recent call last):
...
InvalidToken: ('zope', (<type 'int'>, <type 'long'>), '') in z y
>>> gg.fromUnicode("y y")
[42, 42]
```

class zope.configuration.fields.**Path**(**args*, ***kw*)
A file path name, which may be input as a relative path

Input paths are converted to absolute paths and normalized.

Let's look at an example:

First, we need a "context" for the field that has a path function for converting relative path to an absolute path.

We'll be careful to do this in an os-independent fashion.

```
>>> from zope.configuration.fields import Path
>>> class FauxContext(object):
...    def path(self, p):
...       return os.path.join(os.sep, 'faux', 'context', p)
>>> context = FauxContext()
>>> field = Path().bind(context)
```

Lets try an absolute path first:

```
>>> import os
>>> p = unicode(os.path.join(os.sep, 'a', 'b'))
>>> n = field.fromUnicode(p)
>>> n.split(os.sep)
[u'', u'a', u'b']
```

This should also work with extra spaces around the path:

```
>>> p = "   \n   %s   \n\n   " % p
>>> n = field.fromUnicode(p)
>>> n.split(os.sep)
[u'', u'a', u'b']
```

Now try a relative path:

```
>>> p = unicode(os.path.join('a', 'b'))
>>> n = field.fromUnicode(p)
>>> n.split(os.sep)
[u'', u'faux', u'context', u'a', u'b']
```

class zope.configuration.fields.**Bool**(*title=u''*, *description=u''*, *__name__=''*, *required=True*, *readonly=False*, *constraint=None*, *default=None*, *defaultFactory=None*, *missing_value=<object object at 0x7f0bc9948c70>*)

A boolean value

Values may be input (in upper or lower case) as any of: yes, no, y, n, true, false, t, or f.

```
>>> from zope.configuration.fields import Bool
>>> Bool().fromUnicode(u"yes")
True
>>> Bool().fromUnicode(u"y")
True
>>> Bool().fromUnicode(u"true")
True
>>> Bool().fromUnicode(u"no")
False
```

class zope.configuration.fields.**MessageID**(**args*, ***kw*)

Text string that should be translated.

When a string is converted to a message ID, it is also recorded in the context.

```
>>> from zope.configuration.fields import MessageID
>>> class Info(object):
...     file = 'file location'
...     line = 8
>>> class FauxContext(object):
...     i18n_strings = {}
...     info = Info()
>>> context = FauxContext()
>>> field = MessageID().bind(context)
```

There is a fallback domain when no domain has been specified.

Exchange the warn function so we can make test whether the warning has been issued

```
>>> warned = None
>>> def fakewarn(*args, **kw): #* syntax highlighting
...     global warned
...     warned = args
```

```
>>> import warnings
>>> realwarn = warnings.warn
>>> warnings.warn = fakewarn

>>> i = field.fromUnicode(u"Hello world!")
>>> i
u'Hello world!'
>>> i.domain
'untranslated'
>>> warned
("You did not specify an i18n translation domain for the '' field in file location",)

>>> warnings.warn = realwarn
```

With the domain specified:

```
>>> context.i18n_strings = {}
>>> context.i18n_domain = 'testing'
```

We can get a message id:

```
>>> i = field.fromUnicode(u"Hello world!")
>>> i
u'Hello world!'
>>> i.domain
'testing'
```

In addition, the string has been registered with the context:

```
>>> context.i18n_strings
{'testing': {u'Hello world!': [('file location', 8)]}}

>>> i = field.fromUnicode(u"Foo Bar")
>>> i = field.fromUnicode(u"Hello world!")
>>> from pprint import PrettyPrinter
>>> pprint=PrettyPrinter(width=70).pprint
>>> pprint(context.i18n_strings)
{'testing': {u'Foo Bar': [('file location', 8)],
             u'Hello world!': [('file location', 8),
                               ('file location', 8)]}}

>>> from zope.i18nmessageid import Message
>>> isinstance(context.i18n_strings['testing'].keys()[0], Message)
True
```

Explicit Message IDs

```
>>> i = field.fromUnicode(u'[View-Permission] View')
>>> i
u'View-Permission'
>>> i.default
u'View'

>>> i = field.fromUnicode(u'[] [Some] text')
>>> i
u'[Some] text'
>>> i.default is None
True
```

2.5 zope.configuration.interfaces

class zope.configuration.interfaces.**InvalidToken**
Invaid token in list.

interface zope.configuration.interfaces.**IConfigurationContext**
Configuration Context

The configuration context manages information about the state of the configuration system, such as the package containing the configuration file. More importantly, it provides methods for importing objects and opening files relative to the package.

package
The current package name

This is the name of the package containing the configuration file being executed. If the configuration file was not included by package, then this is None.

resolve(*dottedname*)
Resolve a dotted name to an object

A dotted name is constructed by concatenating a dotted module name with a global name within the module using a dot. For example, the object named "spam" in the foo.bar module has a dotted name of foo.bar.spam. If the current package is a prefix of a dotted name, then the package name can be relaced with a leading dot, So, for example, if the configuration file is in the foo package, then the dotted name foo.bar.spam can be shortened to .bar.spam.

If the current package is multiple levels deep, multiple leading dots can be used to refer to higher-level modules. For example, if the current package is x.y.z, the dotted object name ..foo refers to x.y.foo.

path(*filename*)
Compute a full file name for the given file

If the filename is relative to the package, then the returned name will include the package path, otherwise, the original file name is returned.

checkDuplicate(*filename*)
Check for duplicate imports of the same file.

Raises an exception if this file had been processed before. This is better than an unlimited number of conflict errors.

processFile(*filename*)
Check whether a file needs to be processed.

Return True if processing is needed and False otherwise. If the file needs to be processed, it will be marked as processed, assuming that the caller will procces the file if it needs to be procssed.

action(*discriminator*, *callable*, *args=()*, *kw={}*, *order=0*, *includepath=None*, *info=None*)
Record a configuration action

The job of most directives is to compute actions for later processing. The action method is used to record those actions. The discriminator is used to to find actions that conflict. Actions conflict if they have the same discriminator. The exception to this is the special case of the discriminator with the value None. An actions with a discriminator of None never conflicts with other actions. This is possible to add an order argument to crudely control the order of execution. 'info' is optional source line information, 'includepath' is None (the default) or a tuple of include paths for this action.

provideFeature(*name*)
Record that a named feature is available in this context.

hasFeature(*name*)
Check whether a named feature is available in this context.

interface zope.configuration.interfaces.**IGroupingContext**

before()
Do something before processing nested directives

after()
Do something after processing nested directives

2.6 zope.configuration.name

zope.configuration.name.**resolve**(*name*, *package='zopeproducts'*, *_silly=('__doc__',)*, *_globals={}*)

zope.configuration.name.**getNormalizedName**(*name*, *package*)

zope.configuration.name.**path**(*file=''*, *package='zopeproducts'*, *_silly=('__doc__',)*, *_globals={}*)

2.7 zope.configuration.xmlconfig

class zope.configuration.xmlconfig.**ZopeXMLConfigurationError**(*info*, *etype*, *evalue*)
Zope XML Configuration error

These errors are wrappers for other errors. They include configuration info and the wrapped error type and value.

Example

```
>>> from zope.configuration.xmlconfig import ZopeXMLConfigurationError
>>> v = ZopeXMLConfigurationError("blah", AttributeError, "xxx")
>>> print v
'blah'
    AttributeError: xxx
```

class zope.configuration.xmlconfig.**ZopeSAXParseException**(*v*)
Sax Parser errors, reformatted in an emacs friendly way

Example

```
>>> from zope.configuration.xmlconfig import ZopeSAXParseException
>>> v = ZopeSAXParseException("foo.xml:12:3:Not well formed")
>>> print v
File "foo.xml", line 12.3, Not well formed
```

class zope.configuration.xmlconfig.**ParserInfo**(*file*, *line*, *column*)
Information about a directive based on parser data

This includes the directive location, as well as text data contained in the directive.

Example

```
>>> from zope.configuration.xmlconfig import ParserInfo
>>> info = ParserInfo('tests//sample.zcml', 1, 0)
>>> info
```

```
File "tests//sample.zcml", line 1.0

>>> print info
File "tests//sample.zcml", line 1.0

>>> info.characters("blah\\n")
>>> info.characters("blah")
>>> info.text
u'blah\\nblah'

>>> info.end(7, 0)
>>> info
File "tests//sample.zcml", line 1.0-7.0

>>> print info
File "tests//sample.zcml", line 1.0-7.0
  <configure xmlns='http://namespaces.zope.org/zope'>
    <!-- zope.configure -->
    <directives namespace="http://namespaces.zope.org/zope">
      <directive name="hook" attributes="name implementation module"
         handler="zope.configuration.metaconfigure.hook" />
    </directives>
  </configure>
```

class zope.configuration.xmlconfig.**ConfigurationHandler**(*context*, *testing=False*)

Interface to the xml parser

Translate parser events into calls into the configuration system.

evaluateCondition(*expression*)

Evaluate a ZCML condition.

`expression` is a string of the form "verb arguments".

Currently the supported verbs are `have`, `not-have`, `installed` and `not-installed`.

The `have` and `not-have` verbs each take one argument: the name of a feature:

```
>>> from zope.configuration.config import ConfigurationContext
>>> from zope.configuration.xmlconfig import ConfigurationHandler
>>> context = ConfigurationContext()
>>> context.provideFeature('apidoc')
>>> c = ConfigurationHandler(context, testing=True)
>>> c.evaluateCondition("have apidoc")
True
>>> c.evaluateCondition("not-have apidoc")
False
>>> c.evaluateCondition("have onlinehelp")
False
>>> c.evaluateCondition("not-have onlinehelp")
True
```

Ill-formed expressions raise an error:

```
>>> c.evaluateCondition("want apidoc")
Traceback (most recent call last):
  ...
ValueError: Invalid ZCML condition: 'want apidoc'

>>> c.evaluateCondition("have x y")
```

```
Traceback (most recent call last):
  ...
ValueError: Only one feature allowed: 'have x y'

>>> c.evaluateCondition("have")
Traceback (most recent call last):
  ...
ValueError: Feature name missing: 'have'
```

The `installed` and `not-installed` verbs each take one argument: the dotted name of a pacakge.

If the pacakge is found, in other words, can be imported, then the condition will return true / false:

```
>>> context = ConfigurationContext()
>>> c = ConfigurationHandler(context, testing=True)
>>> c.evaluateCondition('installed zope.interface')
True
>>> c.evaluateCondition('not-installed zope.interface')
False
>>> c.evaluateCondition('installed zope.foo')
False
>>> c.evaluateCondition('not-installed zope.foo')
True
```

Ill-formed expressions raise an error:

```
>>> c.evaluateCondition("installed foo bar")
Traceback (most recent call last):
  ...
ValueError: Only one package allowed: 'installed foo bar'

>>> c.evaluateCondition("installed")
Traceback (most recent call last):
  ...
ValueError: Package name missing: 'installed'
```

zope.configuration.xmlconfig.**processxmlfile**(*file*, *context*, *testing=False*)

Process a configuration file

See examples in tests/test_xmlconfig.py

zope.configuration.xmlconfig.**openInOrPlain**(*filename*)

Open a file, falling back to filename.in.

If the requested file does not exist and filename.in does, fall back to filename.in. If opening the original filename fails for any other reason, allow the failure to propogate.

For example, the tests/samplepackage direxTory has files:

- configure.zcml
- configure.zcml.in
- foo.zcml.in

If we open configure.zcml, we'll get that file:

```
>>> import os
>>> from zope.configuration.xmlconfig import __file__
>>> from zope.configuration.xmlconfig import openInOrPlain
>>> here = os.path.dirname(__file__)
>>> path = os.path.join(here, 'tests', 'samplepackage', 'configure.zcml')
```

```
>>> f = openInOrPlain(path)
>>> f.name[-14:]
'configure.zcml'
```

But if we open foo.zcml, we'll get foo.zcml.in, since there isn't a foo.zcml:

```
>>> path = os.path.join(here, 'tests', 'samplepackage', 'foo.zcml')
>>> f = openInOrPlain(path)
>>> f.name[-11:]
'foo.zcml.in'
```

Make sure other IOErrors are re-raised. We need to do this in a try-except block because different errors are raised on Windows and on Linux.

```
>>> try:
...     f = openInOrPlain('.')
... except IOError:
...     print "passed"
... else:
...     print "failed"
passed
```

interface `zope.configuration.xmlconfig.`**IInclude**

The `include`, `includeOverrides` and `exclude` directives

These directives allows you to include or preserve including of another ZCML file in the configuration. This enables you to write configuration files in each package and then link them together.

file

Configuration file name

The name of a configuration file to be included/excluded, relative to the directive containing the including configuration file.

files

Configuration file name pattern

The names of multiple configuration files to be included/excluded, expressed as a file-name pattern, relative to the directive containing the including or excluding configuration file. The pattern can include:

- `*` matches 0 or more characters
- `?` matches a single character
- `[<seq>]` matches any character in seq
- `[!<seq>]` matches any character not in seq

The file names are included in sorted order, where sorting is without regard to case.

package

Include or exclude package

Include or exclude the named file (or configure.zcml) from the directory of this package.

`zope.configuration.xmlconfig.`**include**(*_context*, *file=None*, *package=None*, *files=None*)

Include a zcml file

See examples in tests/text_xmlconfig.py

`zope.configuration.xmlconfig.`**exclude**(*_context*, *file=None*, *package=None*, *files=None*)

Exclude a zcml file

This directive should be used before any ZML that includes configuration you want to exclude.

zope.configuration.xmlconfig.**includeOverrides**(*_context*, *file=None*, *package=None*, *files=None*)
Include zcml file containing overrides

The actions in the included file are added to the context as if they were in the including file directly.

See the detailed example in test_includeOverrides in tests/text_xmlconfig.py

zope.configuration.xmlconfig.**registerCommonDirectives**(*context*)

zope.configuration.xmlconfig.**file**(*name*, *package=None*, *context=None*, *execute=True*)
Execute a zcml file

zope.configuration.xmlconfig.**string**(*s*, *context=None*, *name='<string>'*, *execute=True*)
Execute a zcml string

class zope.configuration.xmlconfig.**XMLConfig**(*file_name*, *module=None*)
Provide high-level handling of configuration files.

See examples in tests/text_xmlconfig.py

zope.configuration.xmlconfig.**xmlconfig**(*file*, *testing=False*)

zope.configuration.xmlconfig.**testxmlconfig**(*file*)
xmlconfig that doesn't raise configuration errors

This is useful for testing, as it doesn't mask exception types.

2.8 zope.configuration.zopeconfigure

Zope configure directive

This file contains the implementation of the Zope configure directive. It is broken out in a separate file to provide an example of a grouping directive.

The zope configuration directive is a pure grouping directive. It doesn't compute any actions on it's own. Instead, it allows a package to be specified, affecting the interpretation of relative dotted names and file paths. It also allows an i18n domain to be specified. The information collected is used by subdirectives.

To define a grouping directive, we need to do three things:

- Define a schema for the parameters passed to the directive
- Define a handler class.
- Register the class

The parameter schema is given by IZopeConfigure. It specifies a package parameter and an i18n_domain parameter. The package parameter is specified as a `GlobalObject`. This means it must be given as a dotted name that can be resolved through import. The i18n domain is just a plain (not unicode) string.

The handler class has a constructor that takes a context to be adapted and zero or more arguments (depending on the paramter schema). The handler class must implement `zope.configuration.interfaces.IGroupingContext`, which defines hooks `before` and `after`, that are called with no arguments before and after nested directives are processed. If a grouping directive handler creates any actions, or does any computation, this is normally done in either the `before` or `after` hooks. Grouping handlers are normally decorators.

The base class, `zope.configuration.config.GroupingContextDecorator`, is normally used to define grouping directive handlers. It provides:

- An implementation of IConfigurationContext, which grouping directive handlers should normally implement,

- A default implementation of `IGroupingContext` that provides empty hooks.
- Decorator support that uses a `__getattr__` method to delegate attribute accesses to adapted contexts, and
- A constructor that sets the `context` attribute to the adapted context and assigns keyword arguments to attributes.

The `ZopeConfigure` provides handling for the `configure` directive. It subclasses GroupingContextDecorator, and overrides the constructor to set the `basepath` attribute if a `package` argument is provided. Note that it delegates the job of assigning paramters to attribute to the `GroupingContextDecorator` constructor.

The last step is to register the directive using the meta configuration directive. If we wanted to register the Zope `configure` directive for the `zope` namespace, we'd use a meta-configuration directive like:

```
<meta:groupingDirective
   namespace="http://namespaces.zope.org/zope"
   name="configure"
   schema="zope.configuration.zopeconfigure.IZopeConfigure"
   handler="zope.configuration.zopeconfigure.ZopeConfigure"
   >
   Zope configure

   The ``configure`` node is normally used as the root node for a
   configuration file.  It can also be used to specify a package or
   internationalization domain for a group of directives within a
   file by grouping those directives.
</meta:groupingDirective>
```

We use the groupingDirective meta-directive to register a grouping directive. The parameters are self explanatory. The textual contents of the directive provide documentation text, excluding parameter documentation, which is provided by the schema.

(The Zope `configuration` directive is actually registered using a lower-level Python API because it is registered for all namespaces, which isn't supported using the meta-configuration directives.)

interface `zope.configuration.zopeconfigure.`**`IZopeConfigure`**

The `zope:configure` Directive

The zope configuration directive is a pure grouping directive. It doesn't compute any actions on it's own. Instead, it allows a package to be specified, affecting the interpretation of relative dotted names and file paths. It also allows an i18n domain to be specified. The information collected is used by subdirectives.

It may seem that this directive can only be used once per file, but it can be applied whereever it is convenient.

`package`

Package

The package to be used for evaluating relative imports and file names.

`i18n_domain`

Internationalization domain

This is a name for the software project. It must be a legal file-system name as it will be used to contruct names for directories containing translation data. The domain defines a namespace for the message ids used by a project.

class `zope.configuration.zopeconfigure.`**`ZopeConfigure`**(*context*, ***kw*)

Zope configure directive

This file contains the implementation of the Zope configure directive. It is broken out in a separate file to provide an example of a grouping directive.

The zope configuration directive is a pure grouping directive. It doesn't compute any actions on it's own. Instead, it allows a package to be specified, affecting the interpretation of relative dotted names and file paths. It also allows an i18n domain to be specified. The information collected is used by subdirectives.

To define a grouping directive, we need to do three things:

- Define a schema for the parameters passed to the directive
- Define a handler class.
- Register the class

The parameter schema is given by IZopeConfigure. It specifies a package parameter and an i18n_domain parameter. The package parameter is specified as a `GlobalObject`. This means it must be given as a dotted name that can be resolved through import. The i18n domain is just a plain (not unicode) string.

The handler class has a constructor that takes a context to be adapted and zero or more arguments (depending on the paramter schema). The handler class must implement `zope.configuration.interfaces.IGroupingContext`, which defines hooks `before` and `after`, that are called with no arguments before and after nested directives are processed. If a grouping directive handler creates any actions, or does any computation, this is normally done in either the `before` or `after` hooks. Grouping handlers are normally decorators.

The base class, `zope.configuration.config.GroupingContextDecorator`, is normally used to define grouping directive handlers. It provides:

- An implementation of IConfigurationContext, which grouping directive handlers should normally implement,
- A default implementation of `IGroupingContext` that provides empty hooks.
- Decorator support that uses a `__getattr__` method to delegate attribute accesses to adapted contexts, and
- A constructor that sets the `context` attribute to the adapted context and assigns keyword arguments to attributes.

The `ZopeConfigure` provides handling for the `configure` directive. It subclasses GroupingContextDecorator, and overrides the constructor to set the `basepath` attribute if a `package` argument is provided. Note that it delegates the job of assigning paramters to attribute to the `GroupingContextDecorator` constructor.

The last step is to register the directive using the meta configuration directive. If we wanted to register the Zope `configure` directive for the `zope` namespace, we'd use a meta-configuration directive like:

```
<meta:groupingDirective
   namespace="http://namespaces.zope.org/zope"
   name="configure"
   schema="zope.configuration.zopeconfigure.IZopeConfigure"
   handler="zope.configuration.zopeconfigure.ZopeConfigure"
   >
   Zope configure

   The ''configure'' node is normally used as the root node for a
   configuration file.  It can also be used to specify a package or
   internationalization domain for a group of directives within a
   file by grouping those directives.
</meta:groupingDirective>
```

We use the groupingDirective meta-directive to register a grouping directive. The parameters are self explanatory. The textual contents of the directive provide documentation text, excluding parameter documentation, which is provided by the schema.

(The Zope `configuration` directive is actually registered using a lower-level Python API because it is registered for all namespaces, which isn't supported using the meta-configuration directives.)

CHAPTER 3

Hacking on `zope.configuration`

3.1 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
```

3.2 Working in a `virtualenv`

3.2.1 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
```

3.2.2 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
```

3.2.3 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.
```

3.3 Using `zc.buildout`

3.3.1 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'.
```

3.3.2 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.
```

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.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 :)
```

3.5 Contributing to zope.configuration

3.5.1 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.

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.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*

Z

Symbols

A

B

C

D

E

F

G

T

U

W

X

Z