zope.component Documentation

Release 4.3

Zope Foundation Contributors

Jul 12, 2017

Contents

Contents:

CHAPTER 1

Zope Component Architecture

This package, together with *zope.interface*, provides facilities for defining, registering and looking up components. There are two basic kinds of components: adapters and utilities.

Utilities

Utilities are just components that provide an interface and that are looked up by an interface and a name. Let's look at a trivial utility definition:

```
>>> from __future__ import print_function
>>> from zope import interface

>>> class IGreeter(interface.Interface):
...     def greet():
...         "say hello"

>>> @interface.implementer(IGreeter)
... class Greeter(object):
...
...     def __init__(self, other="world"):
...         self.other = other
...
...     def greet(self):
...         print(("Hello %s" % (self.other)))
```

We can register an instance this class using `provideUtility()`[1]:

```
>>> from zope import component
>>> greet = Greeter('bob')
>>> component.provideUtility(greet, IGreeter, 'robert')
```

[1] CAUTION: This API should only be used from test or application-setup code. This API shouldn't be used by regular library modules, as component registration is a configuration activity.

In this example we registered the utility as providing the `IGreeter` interface with a name of 'bob'. We can look the interface up with either *queryUtility()* or *getUtility()*:

```
>>> component.queryUtility(IGreeter, 'robert').greet()
Hello bob

>>> component.getUtility(IGreeter, 'robert').greet()
Hello bob
```

queryUtility() and *getUtility()* differ in how failed lookups are handled:

```
>>> component.queryUtility(IGreeter, 'ted')
>>> component.queryUtility(IGreeter, 'ted', 42)
42
>>> component.getUtility(IGreeter, 'ted')
...
Traceback (most recent call last):
...
ComponentLookupError: (<InterfaceClass ...IGreeter>, 'ted')
```

If a component provides only one interface, as in the example above, then we can omit the provided interface from the call to `provideUtility()`:

```
>>> ted = Greeter('ted')
>>> component.provideUtility(ted, name='ted')
>>> component.queryUtility(IGreeter, 'ted').greet()
Hello ted
```

The name defaults to an empty string:

```
>>> world = Greeter()
>>> component.provideUtility(world)
>>> component.queryUtility(IGreeter).greet()
Hello world
```

Adapters

Adapters are components that are computed from other components to adapt them to some interface. Because they are computed from other objects, they are provided as factories, usually classes. Here, we'll create a greeter for persons, so we can provide personalized greetings for different people:

```
>>> class IPerson(interface.Interface):
...     name = interface.Attribute("Name")

>>> @component.adapter(IPerson)
... @interface.implementer(IGreeter)
... class PersonGreeter(object):
...
...     def __init__(self, person):
...         self.person = person
...
...     def greet(self):
...         print("Hello", self.person.name)
```

The class defines a constructor that takes an argument for every object adapted.

We used *adapter()* to declare what we adapt. We can find out if an object declares that it adapts anything using adaptedBy:

```
>>> list(component.adaptedBy(PersonGreeter)) == [IPerson]
True
```

If an object makes no declaration, then None is returned:

```
>>> component.adaptedBy(Greeter()) is None
True
```

If we declare the interfaces adapted and if we provide only one interface, as in the example above, then we can provide the adapter very simply[1]:

```
>>> component.provideAdapter(PersonGreeter)
```

For adapters that adapt a single interface to a single interface without a name, we can get the adapter by simply calling the interface:

```
>>> @interface.implementer(IPerson)
... class Person(object):
...
...     def __init__(self, name):
...         self.name = name

>>> IGreeter(Person("Sally")).greet()
Hello Sally
```

We can also provide arguments to be very specific about what how to register the adapter.

```
>>> class BobPersonGreeter(PersonGreeter):
...     name = 'Bob'
...     def greet(self):
...         print("Hello", self.person.name, "my name is", self.name)

>>> component.provideAdapter(
...                         BobPersonGreeter, [IPerson], IGreeter, 'bob')
```

The arguments can also be provided as keyword arguments:

```
>>> class TedPersonGreeter(BobPersonGreeter):
...     name = "Ted"

>>> component.provideAdapter(
...     factory=TedPersonGreeter, adapts=[IPerson],
...     provides=IGreeter, name='ted')
```

For named adapters, use *queryAdapter()*, or *getAdapter()*:

```
>>> component.queryAdapter(Person("Sally"), IGreeter, 'bob').greet()
Hello Sally my name is Bob

>>> component.getAdapter(Person("Sally"), IGreeter, 'ted').greet()
Hello Sally my name is Ted
```

If an adapter can't be found, *queryAdapter()* returns a default value and *getAdapter()* raises an error:

```
>>> component.queryAdapter(Person("Sally"), IGreeter, 'frank')
>>> component.queryAdapter(Person("Sally"), IGreeter, 'frank', 42)
42
>>> component.getAdapter(Person("Sally"), IGreeter, 'frank')
...
Traceback (most recent call last):
...
ComponentLookupError: (...Person...>, <...IGreeter>, 'frank')
```

Adapters can adapt multiple objects:

```
>>> @component.adapter(IPerson, IPerson)
... @interface.implementer(IGreeter)
... class TwoPersonGreeter(object):
...
...     def __init__(self, person, greeter):
...         self.person = person
...         self.greeter = greeter
...
...     def greet(self):
...         print("Hello", self.person.name)
...         print("my name is", self.greeter.name)

>>> component.provideAdapter(TwoPersonGreeter)
```

Note that the declaration-order of the Interfaces beeing adapted to is important for adapter look up. It must be the the same as the order of parameters given to the adapter and used to query the adapter. This is especially the case when different Interfaces are adapt to (opposed to this example).

To look up a multi-adapter, use either *queryMultiAdapter()* or *getMultiAdapter()*:

```
>>> component.queryMultiAdapter((Person("Sally"), Person("Bob")),
...                             IGreeter).greet()
Hello Sally
my name is Bob
```

Adapters need not be classes. Any callable will do. We use the adapter decorator to declare that a callable object adapts some interfaces (or classes):

```
>>> class IJob(interface.Interface):
...     "A job"

>>> @interface.implementer(IJob)
... class Job:
...     pass
>>> @interface.implementer(IJob)
... @component.adapter(IPerson)
... def personJob(person):
...     return getattr(person, 'job', None)
```

In this example, the personJob function simply returns the person's `job` attribute if present, or None if it's not present. An adapter factory can return None to indicate that adaptation wasn't possible. Let's register this adapter and try it out:

```
>>> component.provideAdapter(personJob)
>>> sally = Person("Sally")
>>> IJob(sally)
Traceback (most recent call last):
```

```
...
TypeError: ('Could not adapt', ...
```

The adaptation failed because sally didn't have a job. Let's give her one:

```
>>> job = Job()
>>> sally.job = job
>>> IJob(sally) is job
True
```

Subscription Adapters

Unlike regular adapters, subscription adapters are used when we want all of the adapters that adapt an object to a particular adapter.

Consider a validation problem. We have objects and we want to assess whether they meet some sort of standards. We define a validation interface:

```
>>> class IValidate(interface.Interface):
...     def validate(ob):
...         """Determine whether the object is valid
...
...         Return a string describing a validation problem.
...         An empty string is returned to indicate that the
...         object is valid.
...         """
```

Perhaps we have documents:

```
>>> class IDocument(interface.Interface):
...     summary = interface.Attribute("Document summary")
...     body = interface.Attribute("Document text")

>>> @interface.implementer(IDocument)
... class Document(object):
...     def __init__(self, summary, body):
...         self.summary, self.body = summary, body
```

Now, we may want to specify various validation rules for documents. For example, we might require that the summary be a single line:

```
>>> @component.adapter(IDocument)
... @interface.implementer(IValidate)
... class SingleLineSummary(object):
...
...     def __init__(self, doc):
...         self.doc = doc
...
...     def validate(self):
...         if '\n' in self.doc.summary:
...             return 'Summary should only have one line'
...         else:
...             return ''
```

Or we might require the body to be at least 1000 characters in length:

```
>>> @component.adapter(IDocument)
... @interface.implementer(IValidate)
... class AdequateLength(object):
...     def __init__(self, doc):
...         self.doc = doc
...
...     def validate(self):
...         if len(self.doc.body) < 1000:
...             return 'too short'
...         else:
...             return ''
```

We can register these as subscription adapters[1]:

```
>>> component.provideSubscriptionAdapter(SingleLineSummary)
>>> component.provideSubscriptionAdapter(AdequateLength)
```

We can then use the subscribers to validate objects:

```
>>> doc = Document("A\nDocument", "blah")
>>> [adapter.validate()
...  for adapter in component.subscribers([doc], IValidate)
...  if adapter.validate()]
['Summary should only have one line', 'too short']

>>> doc = Document("A\nDocument", "blah" * 1000)
>>> [adapter.validate()
...  for adapter in component.subscribers([doc], IValidate)
...  if adapter.validate()]
['Summary should only have one line']

>>> doc = Document("A Document", "blah")
>>> [adapter.validate()
...  for adapter in component.subscribers([doc], IValidate)
...  if adapter.validate()]
['too short']
```

Handlers

Handlers are subscription adapter factories that don't produce anything. They do all of their work when called. Handlers are typically used to handle events.

Event subscribers are different from other subscription adapters in that the caller of event subscribers doesn't expect to interact with them in any direct way. For example, an event publisher doesn't expect to get any return value. Because subscribers don't need to provide an API to their callers, it is more natural to define them with functions, rather than classes. For example, in a document-management system, we might want to record creation times for documents:

```
>>> import datetime

>>> def documentCreated(event):
...     event.doc.created = datetime.datetime.utcnow()
```

In this example, we have a function that takes an event and performs some processing. It doesn't actually return anything. This is a special case of a subscription adapter that adapts an event to nothing. All of the work is done when

the adapter "factory" is called. We call subscribers that don't actually create anything "handlers". There are special APIs for registering and calling them.

To register the subscriber above, we define a document-created event:

```
>>> class IDocumentCreated(interface.Interface):
...     doc = interface.Attribute("The document that was created")

>>> @interface.implementer(IDocumentCreated)
... class DocumentCreated(object):
...
...     def __init__(self, doc):
...         self.doc = doc
```

We'll also change our handler definition to:

```
>>> @component.adapter(IDocumentCreated)
... def documentCreated(event):
...     event.doc.created = datetime.datetime.utcnow()
```

This marks the handler as an adapter of `IDocumentCreated` events.

Now we'll register the handler[1]:

```
>>> component.provideHandler(documentCreated)
```

Now, if we can create an event and use the *`handle()`* function to call handlers registered for the event:

```
>>> component.handle(DocumentCreated(doc))
>>> doc.created.__class__.__name__
'datetime'
```

CHAPTER 2

The Zope 3 Component Architecture (Socket Example)

The component architecture provides an application framework that provides its functionality through loosely-connected components. A *component* can be any Python object and has a particular purpose associated with it. Thus, in a component-based applications you have many small component in contrast to classical object-oriented development, where you have a few big objects.

Components communicate via specific APIs, which are formally defined by interfaces, which are provided by the *zope.interface* package. *Interfaces* describe the methods and properties that a component is expected to provide. They are also used as a primary mean to provide developer-level documentation for the components. For more details about interfaces see *zope/interface/README.txt*.

The two main types of components are *adapters* and *utilities*. They will be discussed in detail later in this document. Both component types are managed by the *site manager*, with which you can register and access these components. However, most of the site manager's functionality is hidden behind the component architecture's public API, which is documented in *IComponentArchitecture*.

Adapters

Adapters are a well-established pattern. An *adapter* uses an object providing one interface to produce an object that provides another interface. Here an example: Imagine that you purchased an electric shaver in the US, and thus you require the US socket type. You are now traveling in Germany, where another socket style is used. You will need a device, an adapter, that converts from the German to the US socket style.

The functionality of adapters is actually natively provided by the *zope.interface* package and is thus well documented there. The *human.txt* file provides a gentle introduction to adapters, whereby *adapter.txt* is aimed at providing a comprehensive insight into adapters, but is too abstract for many as an initial read. Thus, we will only explain adapters in the context of the component architecture's API.

So let's say that we have a German socket:

```
>>> from zope.interface import Interface, implementer

>>> class IGermanSocket(Interface):
...     pass
```

```
>>> class Socket(object):
...     def __repr__(self):
...         return '<instance of %s>' %self.__class__.__name__

>>> @implementer(IGermanSocket)
... class GermanSocket(Socket):
...     """German wall socket."""
```

and we want to convert it to an US socket

```
>>> class IUSSocket(Interface):
...     pass
```

so that our shaver can be used in Germany. So we go to a German electronics store to look for an adapter that we can plug in the wall:

```
>>> @implementer(IUSSocket)
... class GermanToUSSocketAdapter(Socket):
...     __used_for__ = IGermanSocket
...
...     def __init__(self, socket):
...         self.context = socket
```

Note that I could have called the passed in socket any way I like, but *context* is the standard name accepted.

Single Adapters

Before we can use the adapter, we have to buy it and make it part of our inventory. In the component architecture we do this by registering the adapter with the framework, more specifically with the global site manager:

```
>>> import zope.component
>>> gsm = zope.component.getGlobalSiteManager()
>>> gsm.registerAdapter(GermanToUSSocketAdapter, (IGermanSocket,), IUSSocket)
```

zope.component is the component architecture API that is being presented by this file. You registered an adapter from *IGermanSocket* to *IUSSocket* having no name (thus the empty string).

Anyways, you finally get back to your hotel room and shave, since you have not been able to shave in the plane. In the bathroom you discover a socket:

```
>>> bathroomDE = GermanSocket()
>>> IGermanSocket.providedBy(bathroomDE)
True
```

You now insert the adapter in the German socket

```
>>> bathroomUS = zope.component.getAdapter(bathroomDE, IUSSocket, '')
```

so that the socket now provides the US version:

```
>>> IUSSocket.providedBy(bathroomUS)
True
```

Now you can insert your shaver and get on with your day.

After a week you travel for a couple of days to the Prague and you notice that the Czech have yet another socket type:

```
>>> class ICzechSocket(Interface):
...     pass

>>> @implementer(ICzechSocket)
... class CzechSocket(Socket):
...     pass

>>> czech = CzechSocket()
```

You try to find an adapter for your shaver in your bag, but you fail, since you do not have one:

```
>>> zope.component.getAdapter(czech, IUSSocket, '') \
...
Traceback (most recent call last):
...
ComponentLookupError: (<instance of CzechSocket>,
                       <InterfaceClass __builtin__.IUSSocket>,
                       '')
```

or the more graceful way:

```
>>> marker = object()
>>> socket = zope.component.queryAdapter(czech, IUSSocket, '', marker)
>>> socket is marker
True
```

In the component architecture API any *get** method will fail with a specific exception, if a query failed, whereby methods starting with *query** will always return a *default* value after a failure.

Named Adapters

You are finally back in Germany. You also brought your DVD player and a couple DVDs with you, which you would like to watch. Your shaver was able to convert automatically from 110 volts to 240 volts, but your DVD player cannot. So you have to buy another adapter that also handles converting the voltage and the frequency of the AC current:

```
>>> @implementer(IUSSocket)
... class GermanToUSSocketAdapterAndTransformer(object):
...     __used_for__ = IGermanSocket
...
...     def __init__(self, socket):
...         self.context = socket
```

Now, we need a way to keep the two adapters apart. Thus we register them with a name:

```
>>> gsm.registerAdapter(GermanToUSSocketAdapter,
...                     (IGermanSocket,), IUSSocket, 'shaver',)
>>> gsm.registerAdapter(GermanToUSSocketAdapterAndTransformer,
...                     (IGermanSocket,), IUSSocket, 'dvd')
```

Now we simply look up the adapters using their labels (called *name*):

```
>>> socket = zope.component.getAdapter(bathroomDE, IUSSocket, 'shaver')
>>> socket.__class__ is GermanToUSSocketAdapter
True

>>> socket = zope.component.getAdapter(bathroomDE, IUSSocket, 'dvd')
```

```
>>> socket.__class__ is GermanToUSSocketAdapterAndTransformer
True
```

Clearly, we do not have an adapter for the MP3 player

```
>>> zope.component.getAdapter(bathroomDE, IUSSocket, 'mp3') \
...
Traceback (most recent call last):
...
ComponentLookupError: (<instance of GermanSocket>,
                       <InterfaceClass __builtin__.IUSSocket>,
                       'mp3')
```

but you could use the 'dvd' adapter in this case of course. ;)

Sometimes you want to know all adapters that are available. Let's say you want to know about all the adapters that convert a German to a US socket type:

```
>>> sockets = list(zope.component.getAdapters((bathroomDE,), IUSSocket))
>>> len(sockets)
3
>>> names = sorted([str(name) for name, socket in sockets])
>>> names
['', 'dvd', 'shaver']
```

zope.component.getAdapters() returns a list of tuples. The first entry of the tuple is the name of the adapter and the second is the adapter itself.

Note that the names are always text strings, meaning `unicode` on Python 2:

```
>>> try:
...     text = unicode
... except NameError:
...     text = str
>>> [isinstance(name, text) for name, _ in sockets]
[True, True, True]
```

Multi-Adapters

After watching all the DVDs you brought at least twice, you get tired of them and you want to listen to some music using your MP3 player. But darn, the MP3 player plug has a ground pin and all the adapters you have do not support that:

```
>>> class IUSGroundedSocket(IUSSocket):
...     pass
```

So you go out another time to buy an adapter. This time, however, you do not buy yet another adapter, but a piece that provides the grounding plug:

```
>>> class IGrounder(Interface):
...     pass

>>> @implementer(IGrounder)
... class Grounder(object):
...     def __repr__(self):
...         return '<instance of Grounder>'
```

Then together they will provided a grounded us socket:

```
>>> @implementer(IUSGroundedSocket)
... class GroundedGermanToUSSocketAdapter(object):
...     __used_for__ = (IGermanSocket, IGrounder)
...     def __init__(self, socket, grounder):
...         self.socket, self.grounder = socket, grounder
```

You now register the combination, so that you know you can create a *IUSGroundedSocket*:

```
>>> gsm.registerAdapter(GroundedGermanToUSSocketAdapter,
...                     (IGermanSocket, IGrounder), IUSGroundedSocket, 'mp3')
```

Given the grounder

```
>>> grounder = Grounder()
```

and a German socket

```
>>> livingroom = GermanSocket()
```

we can now get a grounded US socket:

```
>>> socket = zope.component.getMultiAdapter((livingroom, grounder),
...                                         IUSGroundedSocket, 'mp3')
```

```
>>> socket.__class__ is GroundedGermanToUSSocketAdapter
True
>>> socket.socket is livingroom
True
>>> socket.grounder is grounder
True
```

Of course, you do not have a 'dvd' grounded US socket available:

```
>>> zope.component.getMultiAdapter((livingroom, grounder),
...                                IUSGroundedSocket, 'dvd') \
...
Traceback (most recent call last):
...
ComponentLookupError: ((<instance of GermanSocket>,
                        <instance of Grounder>),
                        <InterfaceClass __builtin__.IUSGroundedSocket>,
                        'dvd')
```

```
>>> socket = zope.component.queryMultiAdapter(
...     (livingroom, grounder), IUSGroundedSocket, 'dvd', marker)
>>> socket is marker
True
```

Again, you might want to read *adapter.txt* in *zope.interface* for a more comprehensive coverage of multi-adapters.

Subscribers

While subscribers are directly supported by the adapter registry and are adapters for all theoretical purposes, practically it might be better to think of them as separate components. Subscribers are particularly useful for events.

Let's say one of our adapters overheated and caused a small fire:

```
>>> class IFire(Interface):
...     pass

>>> @implementer(IFire)
... class Fire(object):
...     pass

>>> fire = Fire()
```

We want to use all available objects to put out the fire:

```
>>> class IFireExtinguisher(Interface):
...     def extinguish():
...         pass

>>> from functools import total_ordering
>>> @total_ordering
... class FireExtinguisher(object):
...     def __init__(self, fire):
...         pass
...     def extinguish(self):
...         "Place extinguish code here."
...         print('Used ' + self.__class__.__name__ + '.')
...     def __lt__(self, other):
...         return type(self).__name__ < type(other).__name__
...     def __eq__(self, other):
...         return self is other
```

Here some specific methods to put out the fire:

```
>>> class PowderExtinguisher(FireExtinguisher):
...     pass
>>> gsm.registerSubscriptionAdapter(PowderExtinguisher,
...                                 (IFire,), IFireExtinguisher)

>>> class Blanket(FireExtinguisher):
...     pass
>>> gsm.registerSubscriptionAdapter(Blanket, (IFire,), IFireExtinguisher)

>>> class SprinklerSystem(FireExtinguisher):
...     pass
>>> gsm.registerSubscriptionAdapter(SprinklerSystem,
...                                 (IFire,), IFireExtinguisher)
```

Now let use all these things to put out the fire:

```
>>> extinguishers = zope.component.subscribers((fire,), IFireExtinguisher)
>>> extinguishers.sort()
>>> for extinguisher in extinguishers:
...     extinguisher.extinguish()
Used Blanket.
Used PowderExtinguisher.
Used SprinklerSystem.
```

If no subscribers are found for a particular object, then an empty list is returned:

```
>>> zope.component.subscribers((object(),), IFireExtinguisher)
[]
```

Utilities

Utilities are the second type of component, the component architecture implements. *Utilities* are simply components that provide an interface. When you register an utility, you always register an instance (in contrast to a factory for adapters) since the initialization and setup process of a utility might be complex and is not well defined. In some ways a utility is much more fundamental than an adapter, because an adapter cannot be used without another component, but a utility is always self-contained. I like to think of utilities as the foundation of your application and adapters as components extending beyond this foundation.

Back to our story...

After your vacation is over you fly back home to Tampa, Florida. But it is August now, the middle of the Hurricane season. And, believe it or not, you are worried that you will not be able to shave when the power goes out for several days. (You just hate wet shavers.)

So you decide to go to your favorite hardware store and by a Diesel-powered electric generator. The generator provides of course a US-style socket:

```
>>> @implementer(IUSSocket)
... class Generator(object):
...     def __repr__(self):
...         return '<instance of Generator>'

>>> generator = Generator()
```

Like for adapters, we now have to add the newly-acquired generator to our inventory by registering it as a utility:

```
>>> gsm.registerUtility(generator, IUSSocket)
```

We can now get the utility using

```
>>> utility = zope.component.getUtility(IUSSocket)
>>> utility is generator
True
```

As you can see, it is very simple to register and retrieve utilities. If a utility does not exist for a particular interface, such as the German socket, then the lookup fails

```
>>> zope.component.getUtility(IGermanSocket)
Traceback (most recent call last):
...
ComponentLookupError: (<InterfaceClass __builtin__.IGermanSocket>, '')
```

or more gracefully when specifying a default value:

```
>>> default = object()
>>> utility = zope.component.queryUtility(IGermanSocket, default=default)
>>> utility is default
True
```

Note: The only difference between *getUtility()* and *queryUtility()* is the fact that you can specify a default value for the latter function, so that it will never cause a *ComponentLookupError*.

Named Utilities

It is often desirable to have several utilities providing the same interface per site. This way you can implement any sort of registry using utilities. For this reason, utilities – like adapters – can be named.

In the context of our story, we might want to do the following: You really do not trust gas stations either. What if the roads are blocked after a hurricane and the gas stations run out of oil. So you look for another renewable power source. Then you think about solar panels! After a storm there is usually very nice weather, so why not? Via the Web you order a set of 110V/120W solar panels that provide a regular US-style socket as output:

```
>>> @implementer(IUSSocket)
... class SolarPanel(object):
...     def __repr__(self):
...         return '<instance of Solar Panel>'

>>> panel = SolarPanel()
```

Once it arrives, we add it to our inventory:

```
>>> gsm.registerUtility(panel, IUSSocket, 'Solar Panel')
```

You can now access the solar panel using

```
>>> utility = zope.component.getUtility(IUSSocket, 'Solar Panel')
>>> utility is panel
True
```

Of course, if a utility is not available, then the lookup will simply fail

```
>>> zope.component.getUtility(IUSSocket, 'Wind Mill')
Traceback (most recent call last):
...
ComponentLookupError: (<InterfaceClass __builtin__.IUSSocket>, 'Wind Mill')
```

or more gracefully when specifying a default value:

```
>>> default = object()
>>> utility = zope.component.queryUtility(IUSSocket, 'Wind Mill',
...                                       default=default)
>>> utility is default
True
```

Now you want to look at all the utilities you have for a particular kind. The following API function will return a list of name/utility pairs:

```
>>> utils = sorted(list(zope.component.getUtilitiesFor(IUSSocket)))
>>> [(str(name), socket) for name, socket in utils]
[('', <instance of Generator>), ('Solar Panel', <instance of Solar Panel>)]
```

Another method of looking up all utilities is by using *getAllUtilitiesRegisteredFor(iface)*. This function will return an iterable of utilities (without names); however, it will also return overridden utilities. If you are not using multiple site managers, you will not actually need this method.

```
>>> utils = sorted(list(zope.component.getAllUtilitiesRegisteredFor(IUSSocket)),
...                key=lambda x: type(x).__name__)
>>> utils
[<instance of Generator>, <instance of Solar Panel>]
```

Factories

A *factory* is a special kind of utility that exists to create other components. A factory is always identified by a name. It also provides a title and description and is able to tell the developer what interfaces the created object will provide. The advantage of using a factory to create an object instead of directly instantiating a class or executing any other callable is that we can refer to the factory by name. As long as the name stays fixed, the implementation of the callable can be renamed or moved without a breakage in code.

Let's say that our solar panel comes in parts and they have to be assembled. This assembly would be done by a factory, so let's create one for the solar panel. To do this, we can use a standard implementation of the *IFactory* interface:

```
>>> from zope.component.factory import Factory
>>> factory = Factory(SolarPanel,
...                   'Solar Panel',
...                   'This factory creates a solar panel.')
```

Optionally, I could have also specified the interfaces that the created object will provide, but the factory class is smart enough to determine the implemented interface from the class. We now register the factory:

```
>>> from zope.component.interfaces import IFactory
>>> gsm.registerUtility(factory, IFactory, 'SolarPanel')
```

We can now get a list of interfaces the produced object will provide:

```
>>> ifaces = zope.component.getFactoryInterfaces('SolarPanel')
>>> IUSSocket in ifaces
True
```

By the way, this is equivalent to

```
>>> ifaces2 = factory.getInterfaces()
>>> ifaces is ifaces2
True
```

Of course you can also just create an object:

```
>>> panel = zope.component.createObject('SolarPanel')
>>> panel.__class__ is SolarPanel
True
```

Note: Ignore the first argument (*None*) for now; it is the context of the utility lookup, which is usually an optional argument, but cannot be in this case, since all other arguments beside the *name* are passed in as arguments to the specified callable.

Once you register several factories

```
>>> gsm.registerUtility(Factory(Generator), IFactory, 'Generator')
```

you can also determine, which available factories will create objects providing a certain interface:

```
>>> factories = zope.component.getFactoriesFor(IUSSocket)
>>> factories = sorted([(name, factory.__class__) for name, factory in factories])
>>> [(str(name), kind) for name, kind in factories]
[('Generator', <class 'zope.component.factory.Factory'>), ('SolarPanel', <class 'zope.
↪component.factory.Factory'>)]
```

Site Managers

Why do we need site managers? Why is the component architecture API not sufficient? Some applications, including Zope 3, have a concept of locations. It is often desirable to have different configurations for these location; this can be done by overwriting existing or adding new component registrations. Site managers in locations below the root location, should be able to delegate requests to their parent locations. The root site manager is commonly known as *global site manager*, since it is always available. You can always get the global site manager using the API:

```
>>> gsm = zope.component.getGlobalSiteManager()

>>> from zope.component import globalSiteManager
>>> gsm is globalSiteManager
True
>>> from zope.component.interfaces import IComponentLookup
>>> IComponentLookup.providedBy(gsm)
True
>>> from zope.component.interfaces import IComponents
>>> IComponents.providedBy(gsm)
True
```

You can also lookup at site manager in a given context. The only requirement is that the context can be adapted to a site manager. So let's create a special site manager:

```
>>> from zope.component.globalregistry import BaseGlobalComponents
>>> sm = BaseGlobalComponents()
```

Now we create a context that adapts to the site manager via the *__conform__* method as specified in PEP 246.

```
>>> class Context(object):
...     def __init__(self, sm):
...         self.sm = sm
...     def __conform__(self, interface):
...         if interface.isOrExtends(IComponentLookup):
...             return self.sm
```

We now instantiate the *Context* with our special site manager:

```
>>> context = Context(sm)
>>> context.sm is sm
True
```

We can now ask for the site manager of this context:

```
>>> lsm = zope.component.getSiteManager(context)
>>> lsm is sm
True
```

The site manager instance *lsm* is formally known as a *local site manager* of *context*.

CHAPTER 3

Events

The Component Architecture provides a way to dispatch events to event handlers. Event handlers are registered as *subscribers* a.k.a. *handlers*.

Before we can start we need to import `zope.component.event` to make the dispatching effective:

```
>>> import zope.component.event
```

Consider two event classes:

```
>>> class Event1(object):
...     pass

>>> class Event2(Event1):
...     pass
```

Now consider two handlers for these event classes:

```
>>> called = []

>>> import zope.component
>>> @zope.component.adapter(Event1)
... def handler1(event):
...     called.append(1)

>>> @zope.component.adapter(Event2)
... def handler2(event):
...     called.append(2)
```

We can register them with the Component Architecture:

```
>>> zope.component.provideHandler(handler1)
>>> zope.component.provideHandler(handler2)
```

Now let's go through the events. We'll see that the handlers have been called accordingly:

```
>>> from zope.event import notify
>>> notify(Event1())
>>> called
[1]

>>> del called[:]
>>> notify(Event2())
>>> called.sort()
>>> called
[1, 2]
```

Object events

The `objectEventNotify` function is a subscriber to dispatch ObjectEvents to interested adapters.

First create an object class:

```
>>> class IUseless(zope.interface.Interface):
...     """Useless object"""

>>> @zope.interface.implementer(IUseless)
... class UselessObject(object):
...     """Useless object"""
```

Then create an event class:

```
>>> class IObjectThrownEvent(zope.component.interfaces.IObjectEvent):
...     """An object has been thrown away"""

>>> @zope.interface.implementer(IObjectThrownEvent)
... class ObjectThrownEvent(zope.component.interfaces.ObjectEvent):
...     """An object has been thrown away"""
```

Create an object and an event:

```
>>> hammer = UselessObject()
>>> event = ObjectThrownEvent(hammer)
```

Then notify the event to the subscribers. Since the subscribers list is empty, nothing happens.

```
>>> zope.component.event.objectEventNotify(event)
```

Now create an handler for the event:

```
>>> events = []
>>> def record(*args): #*
...     events.append(args)

>>> zope.component.provideHandler(record, [IUseless, IObjectThrownEvent])
```

The event is notified to the subscriber:

```
>>> zope.component.event.objectEventNotify(event)
>>> events == [(hammer, event)]
True
```

Following test demonstrates how a subscriber can raise an exception to prevent an action.

```
>>> zope.component.provideHandler(zope.component.event.objectEventNotify)
```

Let's create a container:

```
>>> class ToolBox(dict):
...     def __delitem__(self, key):
...         notify(ObjectThrownEvent(self[key]))
...         return super(ToolBox,self).__delitem__(key)

>>> container = ToolBox()
```

And put the object into the container:

```
>>> container['Red Hammer'] = hammer
```

Create an handler function that will raise an error when called:

```
>>> class Veto(Exception):
...     pass

>>> def callback(item, event):
...     assert(item == event.object)
...     raise Veto
```

Register the handler:

```
>>> zope.component.provideHandler(callback, [IUseless, IObjectThrownEvent])
```

Then if we try to remove the object, an ObjectThrownEvent is fired:

```
>>> del container['Red Hammer']
...
Traceback (most recent call last):
...
    raise Veto
Veto
```

CHAPTER 4

Factories

The Factory Class

```
>>> from zope.interface import Interface
>>> class IFunction(Interface):
...     pass

>>> class IKlass(Interface):
...     pass

>>> from zope.interface import implementer
>>> @implementer(IKlass)
... class Klass(object):
...
...     def __init__(self, *args, **kw): #*
...         self.args = args
...         self.kw = kw

>>> from zope.component.factory import Factory
>>> factory = Factory(Klass, 'Klass', 'Klassier')
>>> factory2 = Factory(lambda x: x, 'Func', 'Function')
>>> factory3 = Factory(lambda x: x, 'Func', 'Function', (IFunction,))
```

Calling a Factory

Here we test whether the factory correctly creates the objects and including the correct handling of constructor elements.

First we create a factory that creates instanace of the *Klass* class:

```
>>> factory = Factory(Klass, 'Klass', 'Klassier')
```

Now we use the factory to create the instance

```
>>> kl = factory(1, 2, foo=3)
```

and make sure that the correct class was used to create the object:

```
>>> kl.__class__
<class 'Klass'>
```

Since we passed in a couple positional and a keyword argument

```
>>> kl.args
(1, 2)
>>> kl.kw
{'foo': 3}

>>> factory2(3)
3
>>> factory3(3)
3
```

Title and Description

```
>>> factory.title
'Klass'
>>> factory.description
'Klassier'
>>> factory2.title
'Func'
>>> factory2.description
'Function'
>>> factory3.title
'Func'
>>> factory3.description
'Function'
```

Provided Interfaces

```
>>> implemented = factory.getInterfaces()
>>> implemented.isOrExtends(IKlass)
True
>>> list(implemented) == [IKlass]
True

>>> implemented2 = factory2.getInterfaces()
>>> list(implemented2)
[]

>>> implemented3 = factory3.getInterfaces()
>>> list(implemented3) == [IFunction]
True
```

The Component Architecture Factory API

```
>>> import zope.component
>>> factory = Factory(Klass, 'Klass', 'Klassier')
>>> gsm = zope.component.getGlobalSiteManager()

>>> from zope.component.interfaces import IFactory
>>> gsm.registerUtility(factory, IFactory, 'klass')
```

Creating an Object

```
>>> kl = zope.component.createObject('klass', 1, 2, foo=3)
>>> isinstance(kl, Klass)
True
>>> kl.args
(1, 2)
>>> kl.kw
{'foo': 3}
```

Accessing Provided Interfaces

```
>>> implemented = zope.component.getFactoryInterfaces('klass')
>>> implemented.isOrExtends(IKlass)
True
>>> [iface for iface in implemented] == [IKlass]
True
```

List of All Factories

```
>>> [(str(name), fac.__class__) for name, fac in
...  zope.component.getFactoriesFor(IKlass)]
[('klass', <class 'zope.component.factory.Factory'>)]
```

CHAPTER 5

Persistent Component Management

Persistent component management allows persistent management of components. From a usage point of view, there shouldn't be any new behavior.

CHAPTER 6

ZCML directives

Components may be registered using the registration API exposed by `zope.component` (provideAdapter, provideUtility, etc.). They may also be registered using configuration files. The common way to do that is by using ZCML (Zope Configuration Markup Language), an XML spelling of component registration.

In ZCML, each XML element is a *directive*. There are different top-level directives that let us register components. We will introduce them one by one here.

This helper will let us easily execute ZCML snippets:

```
>>> from io import BytesIO
>>> from zope.configuration.xmlconfig import xmlconfig
>>> def runSnippet(snippet):
...     template = """\
...     <configure xmlns='http://namespaces.zope.org/zope'
...                i18n_domain="zope">
...     %s
...     </configure>"""
...     xmlconfig(BytesIO((template % snippet).encode("ascii")))
```

adapter

Adapters play a key role in the Component Architecture. In ZCML, they are registered with the <adapter /> directive.

```
>>> from zope.component.testfiles.adapter import A1, A2, A3, Handler
>>> from zope.component.testfiles.adapter import I1, I2, I3, IS
>>> from zope.component.testfiles.components import IContent, Content, Comp, comp
```

Before we register the first test adapter, we can verify that adapter lookup doesn't work yet:

```
>>> from zope.component.tests.examples import clearZCML
>>> clearZCML()
>>> from zope.component.testfiles.components import IApp
```

```
>>> IApp(Content(), None) is None
True
```

Then we register the adapter and see that the lookup works:

```
>>> runSnippet('''
...   <adapter
...       factory="zope.component.testfiles.components.Comp"
...       provides="zope.component.testfiles.components.IApp"
...       for="zope.component.testfiles.components.IContent"
...       />''')

>>> IApp(Content()).__class__
<class 'zope.component.testfiles.components.Comp'>
```

It is also possible to give adapters names. Then the combination of required interface, provided interface and name makes the adapter lookup unique. The name is supplied using the `name` argument to the <adapter /> directive:

```
>>> import zope.component
>>> from zope.component.tests.examples import clearZCML
>>> clearZCML()
>>> zope.component.queryAdapter(Content(), IApp, 'test') is None
True

>>> runSnippet('''
...   <adapter
...       factory="zope.component.testfiles.components.Comp"
...       provides="zope.component.testfiles.components.IApp"
...       for="zope.component.testfiles.components.IContent"
...       name="test"
...       />''')

>>> zope.component.getAdapter(Content(), IApp, 'test').__class__
<class 'zope.component.testfiles.components.Comp'>
```

Adapter factories

It is possible to supply more than one adapter factory. In this case, during adapter lookup each factory will be called and the return value will be given to the next factory. The return value of the last factory is returned as the result of the adapter lookup. For examle:

```
>>> clearZCML()
>>> runSnippet('''
...   <adapter
...       factory="zope.component.testfiles.adapter.A1
...                zope.component.testfiles.adapter.A2
...                zope.component.testfiles.adapter.A3"
...       provides="zope.component.testfiles.components.IApp"
...       for="zope.component.testfiles.components.IContent"
...       />''')
```

The resulting adapter is an A3, around an A2, around an A1, around the adapted object:

```
>>> content = Content()
>>> a3 = IApp(content)
>>> a3.__class__ is A3
```

```
True

>>> a2 = a3.context[0]
>>> a2.__class__ is A2
True

>>> a1 = a2.context[0]
>>> a1.__class__ is A1
True

>>> a1.context[0] is content
True
```

Of course, if no factory is provided at all, we will get an error:

```
>>> runSnippet('''
...   <adapter
...       factory=""
...       provides="zope.component.testfiles.components.IApp"
...       for="zope.component.testfiles.components.IContent"
...       />''')
Traceback (most recent call last):
   ...
ZopeXMLConfigurationError: File "<string>", line 4.2-8.8
      ValueError: No factory specified
```

Declaring `for`, `provides` and `name` in Python

The <adapter /> directive can figure out from the in-line Python declaration (using `zope.component.adapts()` or `zope.component.adapter()`, `zope.interface.implements` as well as `zope.component.named`) what the adapter should be registered for and what it provides:

```
>>> clearZCML()
>>> IApp(Content(), None) is None
True

>>> runSnippet('''
...   <adapter factory="zope.component.testfiles.components.Comp" />''')

>>> IApp(Content()).__class__
<class 'zope.component.testfiles.components.Comp'>
```

Of course, if the adapter has no `implements()` declaration, ZCML can't figure out what it provides:

```
>>> runSnippet('''
...   <adapter
...       factory="zope.component.testfiles.adapter.A4"
...       for="zope.component.testfiles.components.IContent"
...       />''')
Traceback (most recent call last):
   ...
ZopeXMLConfigurationError: File "<string>", line 4.2-7.8
      TypeError: Missing 'provides' attribute
```

On the other hand, if the factory implements more than one interface, ZCML can't figure out what it should provide either:

```
>>> runSnippet('''
...   <adapter
...       factory="zope.component.testfiles.adapter.A5"
...       for="zope.component.testfiles.components.IContent"
...       />''')
Traceback (most recent call last):
   ...
ZopeXMLConfigurationError: File "<string>", line 4.2-7.8
      TypeError: Missing 'provides' attribute
```

Let's now register an adapter that has a name specified in Python:

```
>>> runSnippet('''
...   <adapter factory="zope.component.testfiles.components.Comp4" />''')
```

```
>>> zope.component.getAdapter(Content(), IApp, 'app').__class__
<class 'zope.component.testfiles.components.Comp4'>
```

A not so common edge case is registering adapters directly for classes, not for interfaces. For example:

```
>>> clearZCML()
>>> runSnippet('''
...   <adapter
...       for="zope.component.testfiles.components.Content"
...       provides="zope.component.testfiles.adapter.I1"
...       factory="zope.component.testfiles.adapter.A1"
...       />''')

>>> content = Content()
>>> a1 = zope.component.getAdapter(content, I1, '')
>>> isinstance(a1, A1)
True
```

This time, any object providing `IContent` won't work if it's not an instance of the `Content` class:

```
>>> import zope.interface
>>> @zope.interface.implementer(IContent)
... class MyContent(object):
...     pass

>>> zope.component.getAdapter(MyContent(), I1, '')
Traceback (most recent call last):
   ...
ComponentLookupError: ...
```

Multi-adapters

Conventional adapters adapt one object to provide another interface. Multi-adapters adapt several objects at once:

```
>>> clearZCML()
>>> runSnippet('''
...   <adapter
...       for="zope.component.testfiles.components.IContent
...            zope.component.testfiles.adapter.I1
...            zope.component.testfiles.adapter.I2"
...       provides="zope.component.testfiles.adapter.I3"
```

```
...       factory="zope.component.testfiles.adapter.A3"
...       />''')

>>> content = Content()
>>> a1 = A1()
>>> a2 = A2()
>>> a3 = zope.component.queryMultiAdapter((content, a1, a2), I3)
>>> a3.__class__ is A3
True
>>> a3.context == (content, a1, a2)
True
```

You can even adapt an empty list of objects (we call this a null-adapter):

```
>>> clearZCML()
>>> runSnippet('''
...   <adapter
...       for=""
...       provides="zope.component.testfiles.adapter.I3"
...       factory="zope.component.testfiles.adapter.A3"
...       />''')

>>> a3 = zope.component.queryMultiAdapter((), I3)
>>> a3.__class__ is A3
True
>>> a3.context == ()
True
```

Even with multi-adapters, ZCML can figure out the `for` and `provides` parameters from the Python declarations:

```
>>> clearZCML()
>>> runSnippet('''
...   <adapter factory="zope.component.testfiles.adapter.A3" />''')

>>> a3 = zope.component.queryMultiAdapter((content, a1, a2), I3)
>>> a3.__class__ is A3
True
>>> a3.context == (content, a1, a2)
True
```

Chained factories are not supported for multi-adapters, though:

```
>>> clearZCML()
>>> runSnippet('''
...   <adapter
...       for="zope.component.testfiles.components.IContent
...            zope.component.testfiles.adapter.I1
...            zope.component.testfiles.adapter.I2"
...       provides="zope.component.testfiles.components.IApp"
...       factory="zope.component.testfiles.adapter.A1
...                zope.component.testfiles.adapter.A2"
...       />''')
Traceback (most recent call last):
   ...
ZopeXMLConfigurationError: File "<string>", line 4.2-11.8
      ValueError: Can't use multiple factories and multiple for
```

And neither for null-adapters:

```
>>> clearZCML()
>>> runSnippet('''
...   <adapter
...       for=""
...       provides="zope.component.testfiles.components.IApp"
...       factory="zope.component.testfiles.adapter.A1
...                zope.component.testfiles.adapter.A2"
...       />''')
Traceback (most recent call last):
   ...
ZopeXMLConfigurationError: File "<string>", line 4.2-9.8
      ValueError: Can't use multiple factories and multiple for
```

Protected adapters

Adapters can be protected with a permission. First we have to define a permission for which we'll have to register the <permission /> directive:

```
>>> clearZCML()
>>> IApp(Content(), None) is None
True

>>> import zope.security
>>> from zope.configuration.xmlconfig import XMLConfig
>>> XMLConfig('meta.zcml', zope.security)()
>>> runSnippet('''
...   <permission
...       id="y.x"
...       title="XY"
...       description="Allow XY."
...       />
...   <adapter
...       factory="zope.component.testfiles.components.Comp"
...       provides="zope.component.testfiles.components.IApp"
...       for="zope.component.testfiles.components.IContent"
...       permission="y.x"
...       />''')
```

We see that the adapter is a location proxy now so that the appropriate permissions can be found from the context:

```
>>> IApp(Content()).__class__
<class 'zope.component.testfiles.components.Comp'>
>>> type(IApp(Content()))
<class 'zope.location.location.LocationProxy'>
```

We can also go about it a different way. Let's make a public adapter and wrap the adapter in a security proxy. That often happens when an adapter is turned over to untrusted code:

```
>>> clearZCML()
>>> IApp(Content(), None) is None
True

>>> runSnippet('''
...   <adapter
...       factory="zope.component.testfiles.components.Comp"
...       provides="zope.component.testfiles.components.IApp"
```

```
...       for="zope.component.testfiles.components.IContent"
...       permission="zope.Public"
...       />''')

>>> from zope.security.checker import ProxyFactory
>>> adapter = ProxyFactory(IApp(Content()))
>>> from zope.security.proxy import getTestProxyItems
>>> items = [item[0] for item in getTestProxyItems(adapter)]
>>> items
['a', 'f']

>>> from zope.security.proxy import removeSecurityProxy
>>> removeSecurityProxy(adapter).__class__ is Comp
True
```

Of course, this still works when we let the ZCML directive handler figure out `for` and `provides` from the Python declarations:

```
>>> clearZCML()
>>> runSnippet('''
...   <adapter
...       factory="zope.component.testfiles.components.Comp"
...       permission="zope.Public"
...       />''')

>>> adapter = ProxyFactory(IApp(Content()))
>>> [item[0] for item in getTestProxyItems(adapter)]
['a', 'f']
>>> removeSecurityProxy(adapter).__class__ is Comp
True
```

It also works with multi adapters:

```
>>> clearZCML()
>>> runSnippet('''
...   <adapter
...       factory="zope.component.testfiles.adapter.A3"
...       provides="zope.component.testfiles.adapter.I3"
...       for="zope.component.testfiles.components.IContent
...            zope.component.testfiles.adapter.I1
...            zope.component.testfiles.adapter.I2"
...       permission="zope.Public"
...       />''')

>>> content = Content()
>>> a1 = A1()
>>> a2 = A2()
>>> a3 = ProxyFactory(zope.component.queryMultiAdapter((content, a1, a2), I3))
>>> a3.__class__ == A3
True
>>> [item[0] for item in getTestProxyItems(a3)]
['f1', 'f2', 'f3']
```

It's probably not worth mentioning, but when we try to protect an adapter with a permission that doesn't exist, we'll obviously get an error:

```
>>> clearZCML()
>>> runSnippet('''
```

```
...    <adapter
...        factory="zope.component.testfiles.components.Comp"
...        provides="zope.component.testfiles.components.IApp"
...        for="zope.component.testfiles.components.IContent"
...        permission="zope.UndefinedPermission"
...        />''')
Traceback (most recent call last):
   ...
ConfigurationExecutionError: exceptions.ValueError: ('Undefined permission id', 'zope.
↪UndefinedPermission')
   in:
   File "<string>", line 4.2-9.8
   Could not read source.
```

Trusted adapters

Trusted adapters are adapters that are trusted to do anything with the objects they are given so that these objects are not security-proxied. They are registered using the `trusted` argument to the <adapter /> directive:

```
>>> clearZCML()
>>> runSnippet('''
...    <adapter
...        for="zope.component.testfiles.components.IContent"
...        provides="zope.component.testfiles.adapter.I1"
...        factory="zope.component.testfiles.adapter.A1"
...        trusted="yes"
...        />''')
```

With an unproxied object, it's business as usual:

```
>>> ob = Content()
>>> type(I1(ob)) is A1
True
```

With a security-proxied object, however, we get a security-proxied adapter:

```
>>> p = ProxyFactory(ob)
>>> a = I1(p)
>>> type(a)
<... 'zope.security...proxy...Proxy...'>
```

While the adapter is security-proxied, the object it adapts is now proxy-free. The adapter has umlimited access to it:

```
>>> a = removeSecurityProxy(a)
>>> type(a) is A1
True
>>> a.context[0] is ob
True
```

We can also protect the trusted adapter with a permission:

```
>>> clearZCML()
>>> XMLConfig('meta.zcml', zope.security)()
>>> runSnippet('''
...    <permission
...        id="y.x"
```

```
...       title="XY"
...       description="Allow XY."
...       />
...   <adapter
...       for="zope.component.testfiles.components.IContent"
...       provides="zope.component.testfiles.adapter.I1"
...       factory="zope.component.testfiles.adapter.A1"
...       permission="y.x"
...       trusted="yes"
...       />''')
```

Again, with an unproxied object, it's business as usual:

```
>>> ob = Content()
>>> type(I1(ob)) is A1
True
```

With a security-proxied object, we again get a security-proxied adapter:

```
>>> p = ProxyFactory(ob)
>>> a = I1(p)
>>> type(a)
<... 'zope.security...proxy...Proxy...'>
```

Since we protected the adapter with a permission, we now encounter a location proxy behind the security proxy:

```
>>> a = removeSecurityProxy(a)
>>> type(a)
<class 'zope.location.location.LocationProxy'>
>>> a.context[0] is ob
True
```

There's one exception to all of this: When you use the public permission (`zope.Public`), there will be no location proxy:

```
>>> clearZCML()
>>> runSnippet('''
...   <adapter
...       for="zope.component.testfiles.components.IContent"
...       provides="zope.component.testfiles.adapter.I1"
...       factory="zope.component.testfiles.adapter.A1"
...       permission="zope.Public"
...       trusted="yes"
...       />''')

>>> ob = Content()
>>> p = ProxyFactory(ob)
>>> a = I1(p)
>>> type(a)
<... 'zope.security...proxy...Proxy...'>

>>> a = removeSecurityProxy(a)
>>> type(a) is A1
True
```

We can also explicitply pass the `locate` argument to make sure we get location proxies:

```
>>> clearZCML()
>>> runSnippet('''
...   <adapter
...       for="zope.component.testfiles.components.IContent"
...       provides="zope.component.testfiles.adapter.I1"
...       factory="zope.component.testfiles.adapter.A1"
...       trusted="yes"
...       locate="yes"
...       />''')

>>> ob = Content()
>>> p = ProxyFactory(ob)
>>> a = I1(p)
>>> type(a)
<... 'zope.security...proxy...Proxy...'>

>>> a = removeSecurityProxy(a)
>>> type(a)
<class 'zope.location.location.LocationProxy'>
```

subscriber

With the <subscriber /> directive you can register subscription adapters or event subscribers with the adapter registry. Consider this very typical example of a <subscriber /> directive:

```
>>> clearZCML()
>>> runSnippet('''
...   <subscriber
...       provides="zope.component.testfiles.adapter.IS"
...       factory="zope.component.testfiles.adapter.A3"
...       for="zope.component.testfiles.components.IContent
...            zope.component.testfiles.adapter.I1"
...       />''')

>>> content = Content()
>>> a1 = A1()

>>> subscribers = zope.component.subscribers((content, a1), IS)
>>> a3 = subscribers[0]
>>> a3.__class__ is A3
True
>>> a3.context == (content, a1)
True
```

Note how ZCML provides some additional information when registering components, such as the ZCML filename and line numbers:

```
>>> sm = zope.component.getSiteManager()
>>> doc = [reg.info for reg in sm.registeredSubscriptionAdapters()
...        if reg.provided is IS][0]
>>> print(doc)
File "<string>", line 4.2-9.8
  Could not read source.
```

The “fun” behind subscription adapters/subscribers is that when several ones are declared for the same for/provides,

they are all found. With regular adapters, the most specific one (and in doubt the one registered last) wins. Consider these two subscribers:

```
>>> clearZCML()
>>> runSnippet('''
...   <subscriber
...       provides="zope.component.testfiles.adapter.IS"
...       factory="zope.component.testfiles.adapter.A3"
...       for="zope.component.testfiles.components.IContent
...            zope.component.testfiles.adapter.I1"
...       />
...   <subscriber
...       provides="zope.component.testfiles.adapter.IS"
...       factory="zope.component.testfiles.adapter.A2"
...       for="zope.component.testfiles.components.IContent
...            zope.component.testfiles.adapter.I1"
...       />''')

>>> subscribers = zope.component.subscribers((content, a1), IS)
>>> len(subscribers)
2
>>> sorted([a.__class__.__name__ for a in subscribers])
['A2', 'A3']
```

Declaring `for` and `provides` in Python

Like the <adapter /> directive, the <subscriber /> directive can figure out from the in-line Python declaration (using `zope.component.adapts()` or `zope.component.adapter()`) what the subscriber should be registered for:

```
>>> clearZCML()
>>> runSnippet('''
...   <subscriber
...       provides="zope.component.testfiles.adapter.IS"
...       factory="zope.component.testfiles.adapter.A3"
...       />''')

>>> content = Content()
>>> a2 = A2()
>>> subscribers = zope.component.subscribers((content, a1, a2), IS)

>>> a3 = subscribers[0]
>>> a3.__class__ is A3
True
>>> a3.context == (content, a1, a2)
True
```

In the same way the directive can figure out what a subscriber provides:

```
>>> clearZCML()
>>> runSnippet('''
...   <subscriber handler="zope.component.testfiles.adapter.A3" />''')

>>> sm = zope.component.getSiteManager()
>>> a3 = sm.adapters.subscriptions((IContent, I1, I2), None)[0]
>>> a3 is A3
True
```

A not so common edge case is declaring subscribers directly for classes, not for interfaces. For example:

```
>>> clearZCML()
>>> runSnippet('''
...   <subscriber
...       for="zope.component.testfiles.components.Content"
...       provides="zope.component.testfiles.adapter.I1"
...       factory="zope.component.testfiles.adapter.A1"
...       />''')

>>> subs = list(zope.component.subscribers((Content(),), I1))
>>> isinstance(subs[0], A1)
True
```

This time, any object providing `IContent` won't work if it's not an instance of the `Content` class:

```
>>> list(zope.component.subscribers((MyContent(),), I1))
[]
```

Protected subscribers

Subscribers can also be protected with a permission. First we have to define a permission for which we'll have to register the <permission /> directive:

```
>>> clearZCML()
>>> XMLConfig('meta.zcml', zope.security)()
>>> runSnippet('''
...   <permission
...       id="y.x"
...       title="XY"
...       description="Allow XY."
...       />
...   <subscriber
...       provides="zope.component.testfiles.adapter.IS"
...       factory="zope.component.testfiles.adapter.A3"
...       for="zope.component.testfiles.components.IContent
...            zope.component.testfiles.adapter.I1"
...       permission="y.x"
...       />''')

>>> subscribers = zope.component.subscribers((content, a1), IS)
>>> a3 = subscribers[0]
>>> a3.__class__ is A3
True
>>> type(a3)
<class 'zope.location.location.LocationProxy'>
>>> a3.context == (content, a1)
True
```

Trusted subscribers

Like trusted adapters, trusted subscribers are subscribers that are trusted to do anything with the objects they are given so that these objects are not security-proxied. In analogy to the <adapter /> directive, they are registered using the `trusted` argument to the <subscriber /> directive:

```
>>> clearZCML()
>>> runSnippet('''
...   <subscriber
...       provides="zope.component.testfiles.adapter.IS"
...       factory="zope.component.testfiles.adapter.A3"
...       for="zope.component.testfiles.components.IContent
...            zope.component.testfiles.adapter.I1"
...       trusted="yes"
...       />''')
```

With an unproxied object, it's business as usual:

```
>>> subscribers = zope.component.subscribers((content, a1), IS)
>>> a3 = subscribers[0]
>>> a3.__class__ is A3
True
>>> a3.context == (content, a1)
True
>>> type(a3) is A3
True
```

Now with a proxied object. We will see that the subscriber has unproxied access to it, but the subscriber itself is proxied:

```
>>> p = ProxyFactory(content)
>>> a3 = zope.component.subscribers((p, a1), IS)[0]
>>> type(a3)
<... 'zope.security...proxy...Proxy...'>
```

There's no location proxy behind the security proxy:

```
>>> removeSecurityProxy(a3).context[0] is content
True
>>> type(removeSecurityProxy(a3)) is A3
True
```

If you want the trusted subscriber to be located, you'll also have to use the `locate` argument:

```
>>> clearZCML()
>>> runSnippet('''
...   <subscriber
...       provides="zope.component.testfiles.adapter.IS"
...       factory="zope.component.testfiles.adapter.A3"
...       for="zope.component.testfiles.components.IContent
...            zope.component.testfiles.adapter.I1"
...       trusted="yes"
...       locate="yes"
...       />''')
```

Again, it's business as usual with an unproxied object:

```
>>> subscribers = zope.component.subscribers((content, a1), IS)
>>> a3 = subscribers[0]
>>> a3.__class__ is A3
True
>>> a3.context == (content, a1)
True
```

```
>>> type(a3) is A3
True
```

With a proxied object, we again get a security-proxied subscriber:

```
>>> p = ProxyFactory(content)
>>> a3 = zope.component.subscribers((p, a1), IS)[0]

>>> type(a3)
<... 'zope.security...proxy...Proxy...'>

>>> removeSecurityProxy(a3).context[0] is content
True
```

However, thanks to the `locate` argument, we now have a location proxy behind the security proxy:

```
>>> type(removeSecurityProxy(a3))
<class 'zope.location.location.LocationProxy'>
```

Event subscriber (handlers)

Sometimes, subscribers don't need to be adapters that actually provide anything. It's enough that a callable is called for a certain event.

```
>>> clearZCML()
>>> runSnippet('''
...   <subscriber
...       for="zope.component.testfiles.components.IContent
...            zope.component.testfiles.adapter.I1"
...       handler="zope.component.testfiles.adapter.Handler"
...       />''')
```

In this case, simply getting the subscribers is enough to invoke them:

```
>>> list(zope.component.subscribers((content, a1), None))
[]
>>> content.args == ((a1,),)
True
```

utility

Apart from adapters (and subscription adapters), the Component Architecture knows a second kind of component: utilities. They are registered using the <utility /> directive.

Before we register the first test utility, we can verify that utility lookup doesn't work yet:

```
>>> clearZCML()
>>> zope.component.queryUtility(IApp) is None
True
```

Then we register the utility:

```
>>> runSnippet('''
...   <utility
...       component="zope.component.testfiles.components.comp"
...       provides="zope.component.testfiles.components.IApp"
...       />''')
>>> zope.component.getUtility(IApp) is comp
True
```

Like adapters, utilities can also have names. There can be more than one utility registered for a certain interface, as long as they each have a different name.

First, we make sure that there's no utility yet:

```
>>> clearZCML()
>>> zope.component.queryUtility(IApp, 'test') is None
True
```

Then we register it:

```
>>> runSnippet('''
...   <utility
...       component="zope.component.testfiles.components.comp"
...       provides="zope.component.testfiles.components.IApp"
...       name="test"
...       />''')
>>> zope.component.getUtility(IApp, 'test') is comp
True
```

Utilities can also be registered from a factory. In this case, the ZCML handler calls the factory (without any arguments) and registers the returned value as a utility. Typically, you'd pass a class for the factory:

```
>>> clearZCML()
>>> zope.component.queryUtility(IApp) is None
True

>>> runSnippet('''
...   <utility
...       factory="zope.component.testfiles.components.Comp"
...       provides="zope.component.testfiles.components.IApp"
...       />''')
>>> zope.component.getUtility(IApp).__class__ is Comp
True
```

Declaring `provides` in Python

Like other directives, <utility /> can also figure out which interface a utility provides from the Python declaration:

```
>>> clearZCML()
>>> zope.component.queryUtility(IApp) is None
True

>>> runSnippet('''
...   <utility component="zope.component.testfiles.components.comp" />''')
>>> zope.component.getUtility(IApp) is comp
True
```

It won't work if the component that is to be registered doesn't provide anything:

```
>>> clearZCML()
>>> runSnippet('''
...   <utility component="zope.component.testfiles.adapter.a4" />''')
Traceback (most recent call last):
   ...
ZopeXMLConfigurationError: File "<string>", line 4.2-4.61
      TypeError: Missing 'provides' attribute
```

Or if more than one interface is provided (then the ZCML directive handler doesn't know under which the utility should be registered):

```
>>> clearZCML()
>>> runSnippet('''
...   <utility component="zope.component.testfiles.adapter.a5" />''')
Traceback (most recent call last):
   ...
ZopeXMLConfigurationError: File "<string>", line 4.2-4.61
      TypeError: Missing 'provides' attribute
```

We can repeat the same drill for utility factories:

```
>>> clearZCML()
>>> runSnippet('''
...   <utility factory="zope.component.testfiles.components.Comp" />''')
>>> zope.component.getUtility(IApp).__class__ is Comp
True

>>> runSnippet('''
...   <utility factory="zope.component.testfiles.adapter.A4" />''')
Traceback (most recent call last):
   ...
ZopeXMLConfigurationError: File "<string>", line 4.2-4.59
      TypeError: Missing 'provides' attribute

>>> clearZCML()
>>> runSnippet('''
...   <utility factory="zope.component.testfiles.adapter.A5" />''')
Traceback (most recent call last):
   ...
ZopeXMLConfigurationError: File "<string>", line 4.2-4.59
      TypeError: Missing 'provides' attribute
```

Declaring `name` in Python

Let's now register a utility that has a name specified in Python:

```
>>> runSnippet('''
...   <utility component="zope.component.testfiles.components.comp4" />''')
```

```
>>> from zope.component.testfiles.components import comp4
>>> zope.component.getUtility(IApp, name='app') is comp4
True
```

```
>>> runSnippet('''
...   <utility factory="zope.component.testfiles.components.Comp4" />''')
```

```
>>> zope.component.getUtility(IApp, name='app') is comp4
False
>>> zope.component.getUtility(IApp, name='app').__class__
<class 'zope.component.testfiles.components.Comp4'>
```

Protected utilities

TODO:

```
def testProtectedUtility(self):
    """Test that we can protect a utility.

    Also:
    Check that multiple configurations for the same utility and
    don't interfere.
    """
    self.assertEqual(zope.component.queryUtility(IV), None)
    xmlconfig(StringIO(template % (
        '''
        <permission id="tell.everyone" title="Yay" />
        <utility
          component="zope.component.testfiles.components.comp"
          provides="zope.component.testfiles.components.IApp"
          permission="tell.everyone"
          />
        <permission id="top.secret" title="shhhh" />
        <utility
          component="zope.component.testfiles.components.comp"
          provides="zope.component.testfiles.components.IAppb"
          permission="top.secret"
          />
        '''
        )))

    utility = ProxyFactory(zope.component.getUtility(IApp))
    items = getTestProxyItems(utility)
    self.assertEqual(items, [('a', 'tell.everyone'),
                             ('f', 'tell.everyone')
                             ])
    self.assertEqual(removeSecurityProxy(utility), comp)

def testUtilityUndefinedPermission(self):
    config = StringIO(template % (
         '''
         <utility
          component="zope.component.testfiles.components.comp"
          provides="zope.component.testfiles.components.IApp"
          permission="zope.UndefinedPermission"
          />
        '''
        ))
    self.assertRaises(ValueError, xmlconfig, config,
                      testing=1)
```

interface

The <interface /> directive lets us register an interface. Interfaces are registered as named utilities. We therefore needn't go though all the lookup details again, it is sufficient to see whether the directive handler emits the right actions.

First we provide a stub configuration context:

```
>>> import re, pprint
>>> try:
...     from cStringIO import StringIO
... except ImportError:
...     from io import StringIO
>>> atre = re.compile(' at [0-9a-fA-Fx]+')
>>> class Context(object):
...    actions = ()
...    def action(self, discriminator, callable, args):
...        self.actions += ((discriminator, callable, args), )
...    def __repr__(self):
...        stream = StringIO()
...        pprinter = pprint.PrettyPrinter(stream=stream, width=60)
...        pprinter.pprint(self.actions)
...        r = stream.getvalue()
...        return (u''.join(atre.split(r))).strip()
>>> context = Context()
```

Then we provide a test interface that we'd like to register:

```
>>> from zope.interface import Interface
>>> class I(Interface):
...     pass
```

It doesn't yet provide `ITestType`:

```
>>> from zope.component.tests.examples import ITestType
>>> ITestType.providedBy(I)
False
```

However, after calling the directive handler...

```
>>> from zope.component.zcml import interface
>>> interface(context, I, ITestType)
>>> context
((None,
  <function provideInterface>,
  ('',
   <InterfaceClass ....I>,
   <InterfaceClass zope.component.tests.examples.ITestType>)),)
```

...it does provide `ITestType`:

```
>>> from zope.interface.interfaces import IInterface
>>> ITestType.extends(IInterface)
True
>>> IInterface.providedBy(I)
True
```

CHAPTER 7

Package configuration

The `zope.component` package provides a ZCML file that configures some basic event handlers.

```
>>> from zope.configuration.xmlconfig import XMLConfig
>>> import zope.component
>>> from zope.component import event
>>> from zope.component import registry

>>> XMLConfig('configure.zcml', zope.component)()

>>> gsm = zope.component.getGlobalSiteManager()
>>> registered = list(gsm.registeredHandlers())
>>> len(registered)
5
>>> handlers = [x.handler for x in registered]
>>> event.objectEventNotify in handlers
True
>>> registry.dispatchUtilityRegistrationEvent in handlers
True
>>> registry.dispatchAdapterRegistrationEvent in handlers
True
>>> registry.dispatchSubscriptionAdapterRegistrationEvent in handlers
True
>>> registry.dispatchHandlerRegistrationEvent in handlers
True
```

CHAPTER 8

The current component registry

There can be any number of component registries in an application. One of them is the global component registry, and there is also the concept of a currently used component registry. Component registries other than the global one are associated with objects called sites. The `zope.component.hooks` module provides an API to set and access the current site as well as manipulate the adapter hook associated with it.

As long as we haven't set a site, none is being considered current:

```
>>> from zope.component.hooks import getSite
>>> print(getSite())
None
```

We can also ask for the current component registry (aka site manager historically); it will return the global one if no current site is set:

```
>>> from zope.component.hooks import getSiteManager
>>> getSiteManager()
<BaseGlobalComponents base>
```

Let's set a site now. A site has to be an object that provides the `getSiteManager` method, which is specified by `zope.component.interfaces.IPossibleSite`:

```
>>> from zope.interface.registry import Components
>>> class Site(object):
...     def __init__(self):
...         self.registry = Components('components')
...     def getSiteManager(self):
...         return self.registry

>>> from zope.component.hooks import setSite
>>> site1 = Site()
>>> setSite(site1)
```

After this, the newly set site is considered the currently active one:

```
>>> getSite() is site1
True
>>> getSiteManager() is site1.registry
True
```

If we set another site, that one will be considered current:

```
>>> site2 = Site()
>>> site2.registry is not site1.registry
True
>>> setSite(site2)

>>> getSite() is site2
True
>>> getSiteManager() is site2.registry
True
```

Finally we can unset the site and the global component registry is used again:

```
>>> setSite()
>>> print(getSite())
None
>>> getSiteManager()
<BaseGlobalComponents base>
```

Context manager

There also is a context manager for setting the site, which is especially useful when writing tests:

```
>>> import zope.component.hooks
>>> print(getSite())
None
>>> with zope.component.hooks.site(site2):
...     getSite() is site2
True
>>> print(getSite())
None
```

The site is properly restored even if the body of the with statement raises an exception:

```
>>> print(getSite())
None
>>> with zope.component.hooks.site(site2):
...     getSite() is site2
...     raise ValueError('An error in the body')
Traceback (most recent call last):
   ...
ValueError: An error in the body
>>> print(getSite())
None
```

CHAPTER 9

Layers

zope.component.testlayer defines two things:

- a LayerBase that makes it easier and saner to use zope.testing's test layers.
- a ZCMLLayer which lets you implement a layer that loads up some ZCML.

LayerBase

We check whether our LayerBase can be used to create layers of our own. We do this simply by subclassing:

```
>>> from zope.component.testlayer import LayerBase
>>> class OurLayer(LayerBase):
...     def setUp(self):
...         super(OurLayer, self).setUp()
...         print("setUp called")
...     def tearDown(self):
...         super(OurLayer, self).tearDown()
...         print("tearDown called")
...     def testSetUp(self):
...         super(OurLayer, self).testSetUp()
...         print("testSetUp called")
...     def testTearDown(self):
...         super(OurLayer, self).testTearDown()
...         print("testTearDown called")
```

Note that if we wanted to ensure that the methods of the superclass were called we have to use super(). In this case we actually wouldn't need to, as these methods do nothing at all, but we just ensure that they are there in the first place.

Let's instantiate our layer. We need to supply it with the package the layer is defined in:

```
>>> import zope.component
>>> layer = OurLayer(zope.component)
```

Now we run some tests with this layer:

```
>>> import unittest
>>> class TestCase(unittest.TestCase):
...     layer = layer
...
...     def testFoo(self):
...         print("testFoo")
>>> suite = unittest.TestSuite()
>>> suite.addTest(unittest.makeSuite(TestCase))
>>> from zope.testrunner.runner import Runner
>>> runner = Runner(args=[], found_suites=[suite])
>>> succeeded = runner.run()
Running zope.component.OurLayer tests:
  Set up zope.component.OurLayer setUp called
in ... seconds.
testSetUp called
testFoo
testTearDown called
  Ran 1 tests with 0 failures, 0 errors and 0 skipped in ... seconds.
Tearing down left over layers:
  Tear down zope.component.OurLayer tearDown called
in ... seconds.
```

ZCMLLayer

We now want a layer that loads up some ZCML from a file. The default is `ftesting.zcml`, but here we'll load a test `testlayer.zcml`. We can also choose to provide extra ZCML features that are used to conditionally control processing of certain directives (here we use "devmode", a common condition for controlling development options like debugging output).

```
>>> from zope.component.testlayer import ZCMLFileLayer
>>> import zope.component.testfiles
>>> zcml_file_layer = ZCMLFileLayer(
...     zope.component.testfiles,
...     'testlayer.zcml',
...     features=["devmode"])

>>> class TestCase(unittest.TestCase):
...     layer = zcml_file_layer
...
...     def testFoo(self):
...         # The feature was registered
...         self.assertTrue(self.layer.context.hasFeature('devmode'))
...         # we should now have the adapter registered
...         from zope import component
...         from zope.component.testfiles import components
...         self.assertIsInstance(
...             components.IApp2(components.content), components.Comp2)
```

Since the ZCML sets up an adapter, we expect the tests to pass:

```
>>> suite = unittest.TestSuite()
>>> suite.addTest(unittest.makeSuite(TestCase))
>>> runner = Runner(args=[], found_suites=[suite])
>>> succeeded = runner.run()
```

```
Running zope.component.testfiles.ZCMLFileLayer tests:
  Set up zope.component.testfiles.ZCMLFileLayer in ... seconds.
  Ran 1 tests with 0 failures, 0 errors and 0 skipped in ... seconds.
Tearing down left over layers:
  Tear down zope.component.testfiles.ZCMLFileLayer in ... seconds.
```

CHAPTER 10

zope.component API Reference

Interface Definitions

Component and Component Architecture Interfaces

interface zope.component.interfaces.**IComponentArchitecture**

The Component Architecture is defined by two key components: Adapters and Utiltities. Both are managed by site managers. All other components build on top of them.

getGlobalSiteManager()

Return the global site manager.

This function should never fail and always return an object that provides *IGlobalSiteManager*.

getSiteManager(*context=None*)

Get the nearest site manager in the given context.

If *context* is *None*, return the global site manager.

If the *context* is not *None*, it is expected that an adapter from the *context* to *IComponentLookup* can be found. If no adapter is found, a *ComponentLookupError* is raised.

getUtility(*interface*, *name=''*, *context=None*)

Get the utility that provides interface

Returns the nearest utility to the context that implements the specified interface. If one is not found, raises ComponentLookupError.

queryUtility(*interface*, *name=''*, *default=None*, *context=None*)

Look for the utility that provides interface

Returns the nearest utility to the context that implements the specified interface. If one is not found, returns default.

queryNextUtility(*context*, *interface*, *name=''*, *default=None*)

Query for the next available utility.

Find the next available utility providing *interface* and having the specified name. If no utility was found, return the specified *default* value.

getNextUtility(*context*, *interface*, *name=''*)
Get the next available utility.

If no utility was found, a *ComponentLookupError* is raised.

getUtilitiesFor(*interface*, *context=None*)
Return the utilities that provide an interface

An iterable of utility name-value pairs is returned.

getAllUtilitiesRegisteredFor(*interface*, *context=None*)
Return all registered utilities for an interface

This includes overridden utilities.

An iterable of utility instances is returned. No names are returned.

getAdapter(*object*, *interface=<InterfaceClass zope.interface.Interface>*, *name=u''*, *context=None*)
Get a named adapter to an interface for an object

Returns an adapter that can adapt object to interface. If a matching adapter cannot be found, raises ComponentLookupError.

If context is None, an application-defined policy is used to choose an appropriate service manager from which to get an 'Adapters' service.

If 'context' is not None, context is adapted to IServiceService, and this adapter's 'Adapters' service is used.

getAdapterInContext(*object*, *interface*, *context*)
Get a special adapter to an interface for an object

NOTE: This method should only be used if a custom context needs to be provided to provide custom component lookup. Otherwise, call the interface, as in:

```
interface(object)
```

Returns an adapter that can adapt object to interface. If a matching adapter cannot be found, raises ComponentLookupError.

Context is adapted to IServiceService, and this adapter's 'Adapters' service is used.

If the object has a __conform__ method, this method will be called with the requested interface. If the method returns a non-None value, that value will be returned. Otherwise, if the object already implements the interface, the object will be returned.

getMultiAdapter(*objects*, *interface=<InterfaceClass zope.interface.Interface>*, *name=''*, *context=None*)
Look for a multi-adapter to an interface for an objects

Returns a multi-adapter that can adapt objects to interface. If a matching adapter cannot be found, raises ComponentLookupError.

If context is None, an application-defined policy is used to choose an appropriate service manager from which to get an 'Adapters' service.

If 'context' is not None, context is adapted to IServiceService, and this adapter's 'Adapters' service is used.

The name consisting of an empty string is reserved for unnamed adapters. The unnamed adapter methods will often call the named adapter methods with an empty string for a name.

queryAdapter(*object*, *interface=<InterfaceClass zope.interface.Interface>*, *name=u''*, *default=None*, *context=None*)
Look for a named adapter to an interface for an object

Returns an adapter that can adapt object to interface. If a matching adapter cannot be found, returns the default.

If context is None, an application-defined policy is used to choose an appropriate service manager from which to get an 'Adapters' service.

If 'context' is not None, context is adapted to IServiceService, and this adapter's 'Adapters' service is used.

queryAdapterInContext(*object*, *interface*, *context*, *default=None*)
Look for a special adapter to an interface for an object

NOTE: This method should only be used if a custom context needs to be provided to provide custom component lookup. Otherwise, call the interface, as in:

```
interface(object, default)
```

Returns an adapter that can adapt object to interface. If a matching adapter cannot be found, returns the default.

Context is adapted to IServiceService, and this adapter's 'Adapters' service is used.

If the object has a __conform__ method, this method will be called with the requested interface. If the method returns a non-None value, that value will be returned. Otherwise, if the object already implements the interface, the object will be returned.

queryMultiAdapter(*objects*, *interface=<InterfaceClass zope.interface.Interface>*, *name=u''*, *default=None*, *context=None*)
Look for a multi-adapter to an interface for objects

Returns a multi-adapter that can adapt objects to interface. If a matching adapter cannot be found, returns the default.

If context is None, an application-defined policy is used to choose an appropriate service manager from which to get an 'Adapters' service.

If 'context' is not None, context is adapted to IServiceService, and this adapter's 'Adapters' service is used.

The name consisting of an empty string is reserved for unnamed adapters. The unnamed adapter methods will often call the named adapter methods with an empty string for a name.

getAdapters(*objects*, *provided*, *context=None*)
Look for all matching adapters to a provided interface for objects

Return a list of adapters that match. If an adapter is named, only the most specific adapter of a given name is returned.

If context is None, an application-defined policy is used to choose an appropriate service manager from which to get an 'Adapters' service.

If 'context' is not None, context is adapted to IServiceService, and this adapter's 'Adapters' service is used.

subscribers(*required*, *provided*, *context=None*)
Get subscribers

Subscribers are returned that provide the provided interface and that depend on and are computed from the sequence of required objects.

If context is None, an application-defined policy is used to choose an appropriate service manager from which to get an 'Adapters' service.

If 'context' is not None, context is adapted to IServiceService, and this adapter's 'Adapters' service is used.

handle(**objects*)
Call all of the handlers for the given objects

Handlers are subscription adapter factories that don't produce anything. They do all of their work when called. Handlers are typically used to handle events.

adapts(**interfaces*)
Declare that a class adapts the given interfaces.

This function can only be used in a class definition.

(TODO, allow classes to be passed as well as interfaces.)

createObject(*factory_name*, **args*, ***kwargs*)
Create an object using a factory

Finds the named factory in the current site and calls it with the given arguments. If a matching factory cannot be found raises ComponentLookupError. Returns the created object.

A context keyword argument can be provided to cause the factory to be looked up in a location other than the current site. (Of course, this means that it is impossible to pass a keyword argument named "context" to the factory.

getFactoryInterfaces(*name*, *context=None*)
Get interfaces implemented by a factory

Finds the factory of the given name that is nearest to the context, and returns the interface or interface tuple that object instances created by the named factory will implement.

getFactoriesFor(*interface*, *context=None*)
Return a tuple (name, factory) of registered factories that create objects which implement the given interface.

interface zope.component.interfaces.**IRegistry**
Object that supports component registry

registrations()
Return an iterable of component registrations

interface zope.component.interfaces.**IComponentRegistrationConvenience**
API for registering components.

CAUTION: This API should only be used from test or application-setup code. This api shouldn't be used by regular library modules, as component registration is a configuration activity.

provideUtility(*component*, *provides=None*, *name=u''*)
Register a utility globally

A utility is registered to provide an interface with a name. If a component provides only one interface, then the provides argument can be omitted and the provided interface will be used. (In this case, provides argument can still be provided to provide a less specific interface.)

CAUTION: This API should only be used from test or application-setup code. This API shouldn't be used by regular library modules, as component registration is a configuration activity.

provideAdapter(*factory*, *adapts=None*, *provides=None*, *name=u''*)
Register an adapter globally

An adapter is registered to provide an interface with a name for some number of object types. If a factory implements only one interface, then the provides argument can be omitted and the provided interface will be used. (In this case, a provides argument can still be provided to provide a less specific interface.)

If the factory has an adapts declaration, then the adapts argument can be omitted and the declaration will be used. (An adapts argument can be provided to override the declaration.)

CAUTION: This API should only be used from test or application-setup code. This API shouldn't be used by regular library modules, as component registration is a configuration activity.

provideSubscriptionAdapter(*factory*, *adapts=None*, *provides=None*)
Register a subscription adapter

A subscription adapter is registered to provide an interface for some number of object types. If a factory implements only one interface, then the provides argument can be omitted and the provided interface will be used. (In this case, a provides argument can still be provided to provide a less specific interface.)

If the factory has an adapts declaration, then the adapts argument can be omitted and the declaration will be used. (An adapts argument can be provided to override the declaration.)

CAUTION: This API should only be used from test or application-setup code. This API shouldn't be used by regular library modules, as component registration is a configuration activity.

provideHandler(*handler*, *adapts=None*)
Register a handler

Handlers are subscription adapter factories that don't produce anything. They do all of their work when called. Handlers are typically used to handle events.

If the handler has an adapts declaration, then the adapts argument can be omitted and the declaration will be used. (An adapts argument can be provided to override the declaration.)

CAUTION: This API should only be used from test or application-setup code. This API shouldn't be used by regular library modules, as component registration is a configuration activity.

interface zope.component.interfaces.**IPossibleSite**
An object that could be a site.

setSiteManager(*sitemanager*)
Sets the site manager for this object.

getSiteManager()
Returns the site manager contained in this object.

If there isn't a site manager, raise a component lookup.

interface zope.component.interfaces.**ISite**
Extends: *zope.component.interfaces.IPossibleSite*

Marker interface to indicate that we have a site

exception zope.component.interfaces.**Misused**
A component is being used (registered) for the wrong interface.

interface zope.component.interfaces.**IFactory**
A factory is responsible for creating other components.

title
The factory title.

description
A brief description of the factory.

__call__(**args*, ***kw*)
Return an instance of the objects we're a factory for.

getInterfaces()
Get the interfaces implemented by the factory

Return the interface(s), as an instance of Implements, that objects created by this factory will implement. If the callable's Implements instance cannot be created, an empty Implements instance is returned.

Site Manager APIs

zope.component.**getGlobalSiteManager**()
The API returns the module-scope global registry:

```
>>> from zope.component.interfaces import IComponentLookup
>>> from zope.component.globalregistry import base
>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()
>>> gsm is base
True
```

The registry implements the IComponentLookup interface:

```
>>> IComponentLookup.providedBy(gsm)
True
```

The same registry is returned each time we call the function:

```
>>> getGlobalSiteManager() is gsm
True
```

zope.component.**getSiteManager**(*context=None*)
Callable objects that support being overridden

We don't know anything about the default service manager, except that it is an *IComponentLookup*.

```
>>> from zope.component import getSiteManager
>>> from zope.component.interfaces import IComponentLookup
>>> IComponentLookup.providedBy(getSiteManager())
True
```

Calling *getSiteManager()* with no args is equivalent to calling it with a context of *None*.

```
>>> getSiteManager() is getSiteManager(None)
True
```

If the context passed to *getSiteManager()* is not *None*, it is adapted to *IComponentLookup* and this adapter returned. So, we create a context that can be adapted to *IComponentLookup* using the *__conform__* API.

Let's create the simplest stub-implementation of a site manager possible:

```
>>> sitemanager = object()
```

Now create a context that knows how to adapt to our newly created site manager.

```
>>> from zope.component.tests.examples import ConformsToIComponentLookup
>>> context = ConformsToIComponentLookup(sitemanager)
```

Now make sure that the *getSiteManager()* API call returns the correct site manager.

```
>>> getSiteManager(context) is sitemanager
True
```

Using a context that is not adaptable to *IComponentLookup* should fail.

```
>>> getSiteManager(sitemanager)
Traceback (most recent call last):
...
ComponentLookupError: ('Could not adapt', <instance Ob>,
<InterfaceClass zope...interfaces.IComponentLookup>)
```

Utility Registration APIs

`zope.component.`**`getUtility`**(*interface*, *name=''*, *context=None*)

`zope.component.`**`queryUtility`**(*interface*, *name=''*, *default=None*, *context=None*)

Utilities are components that simply provide an interface. They are instantiated at the time or before they are registered. Here we test the simple query interface.

Before we register any utility, there is no utility available, of course. The pure instatiation of an object does not make it a utility. If you do not specify a default, you get a *ComponentLookupError*.

```
>>> from zope.component import getUtility
>>> from zope.component import queryUtility
>>> from zope.component.tests.examples import I1
>>> getUtility(I1)
Traceback (most recent call last):
...
ComponentLookupError: \
(<InterfaceClass zope.component.tests.examples.I1>, '')
```

Otherwise, you get the default:

```
>>> queryUtility(I1, default='<default>')
'<default>'
```

Now we declare *ob* to be the utility providing *I1*:

```
>>> ob = object()
>>> from zope.component import getGlobalSiteManager
>>> getGlobalSiteManager().registerUtility(ob, I1)
```

Now the component is available:

```
>>> getUtility(I1) is ob
True
>>> queryUtility(I1) is ob
True
```

Named Utilities

Registering a utility without a name does not mean that it is available when looking for the utility with a name:

```
>>> getUtility(I1, name='foo')
Traceback (most recent call last):
...
ComponentLookupError:
(<InterfaceClass zope.component.tests.examples.I1>, 'foo')

>>> queryUtility(I1, name='foo', default='<default>')
'<default>'
```

Registering the utility under the correct name makes it available:

```
>>> ob2 = object()
>>> getGlobalSiteManager().registerUtility(ob2, I1, name='foo')
>>> getUtility(I1, 'foo') is ob2
True
>>> queryUtility(I1, 'foo') is ob2
True
```

Querying Multiple Utilities

`zope.component.`**`getUtilitiesFor`**(*interface*, *context=None*)

`zope.component.`**`getAllUtilitiesRegisteredFor`**(*interface*, *context=None*)

Sometimes it may be useful to query all utilities, both anonymous and named for a given interface. The *`getUtilitiesFor()`* API returns a sequence of `(name, utility)` tuples, where `name` is the empty string for the anonymous utility:

```
>>> from zope.component import getUtilitiesFor
>>> tuples = list(getUtilitiesFor(I1))
>>> len(tuples)
2
>>> ('', ob) in tuples
True
>>> ('foo', ob2) in tuples
True
```

The *`getAllUtilitiesRegisteredFor()`* API returns utilities that have been registered for a particular interface. Utilities providing a derived interface are also listed.

```
>>> from zope.interface import implementer
>>> from zope.component.tests.examples import Comp
>>> from zope.component.tests.examples import I2
>>> from zope.component.tests.examples import Ob
>>> class I11(I1):
...     pass

>>> @implementer(I11)
... class Ob11(Ob):
...     pass

>>> ob11 = Ob11()
>>> ob_bob = Ob()
```

Now we register the new utilities:

```
>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()
>>> gsm.registerUtility(ob, I1)
>>> gsm.registerUtility(ob11, I11)
>>> gsm.registerUtility(ob_bob, I1, name='bob')
>>> gsm.registerUtility(Comp(2), I2)
```

We can now get all the utilities that provide interface *I1*:

```
>>> from zope.component import getAllUtilitiesRegisteredFor
>>> uts = list(getAllUtilitiesRegisteredFor(I1))
>>> len(uts)
4
>>> ob in uts
True
>>> ob2 in uts
True
>>> ob_bob in uts
True
>>> ob11 in uts
True
```

Note that *getAllUtilitiesRegisteredFor()* does not return the names of the utilities.

Delegated Utility Lookup

zope.component.**getNextUtility**(*context*, *interface*, *name=''*)
Get the next available utility.

If no utility was found, a *ComponentLookupError* is raised.

zope.component.**queryNextUtility**(*context*, *interface*, *name=''*, *default=None*)
Query for the next available utility.

Find the next available utility providing *interface* and having the specified name. If no utility was found, return the specified *default* value.

It is common for a utility to delegate its answer to a utility providing the same interface in one of the component registry's bases. Let's first create a global utility:

```
>>> from zope.interface import Interface
>>> from zope.interface import implementer
>>> class IMyUtility(Interface):
...     pass

>>> from zope.component.tests.examples import ConformsToIComponentLookup
>>> @implementer(IMyUtility)
... class MyUtility(ConformsToIComponentLookup):
...     def __init__(self, id, sm):
...         self.id = id
...         self.sitemanager = sm
...     def __repr__(self):
...         return "%s('%s')" % (self.__class__.__name__, self.id)

>>> gutil = MyUtility('global', gsm)
>>> gsm.registerUtility(gutil, IMyUtility, 'myutil')
```

Now, let's create two registries and set up the bases hierarchy:

```
>>> from zope.interface.registry import Components
>>> sm1 = Components('sm1', bases=(gsm, ))
>>> sm1_1 = Components('sm1_1', bases=(sm1, ))
```

Now we create two utilities and insert them in our folder hierarchy:

```
>>> from zope.component.interfaces import IComponentLookup
>>> util1 = MyUtility('one', sm1)
>>> sm1.registerUtility(util1, IMyUtility, 'myutil')
>>> IComponentLookup(util1) is sm1
True

>>> util1_1 = MyUtility('one-one', sm1_1)
>>> sm1_1.registerUtility(util1_1, IMyUtility, 'myutil')
>>> IComponentLookup(util1_1) is sm1_1
True
```

Now, if we ask *util1_1* for its next available utility we get the `one` utility:

```
>>> from zope.component import getNextUtility
>>> getNextUtility(util1_1, IMyUtility, 'myutil')
MyUtility('one')
```

Next we ask *util1* for its next utility and we should get the global version:

```
>>> getNextUtility(util1, IMyUtility, 'myutil')
MyUtility('global')
```

However, if we ask the global utility for the next one, an error is raised

```
>>> getNextUtility(gutil, IMyUtility,
...                        'myutil')
Traceback (most recent call last):
...
ComponentLookupError:
No more utilities for <InterfaceClass zope.component.tests.examples.IMyUtility>,
'myutil' have been found.
```

You can also use *queryNextUtility* and specify a default:

```
>>> from zope.component import queryNextUtility
>>> queryNextUtility(gutil, IMyUtility, 'myutil', 'default')
'default'
```

Let's now ensure that the function also works with multiple registries. First we create another base registry:

```
>>> myregistry = Components()
```

We now set up another utility into that registry:

```
>>> custom_util = MyUtility('my_custom_util', myregistry)
>>> myregistry.registerUtility(custom_util, IMyUtility, 'my_custom_util')
```

We add it as a base to the local site manager:

```
>>> sm1.__bases__ = (myregistry,) + sm1.__bases__
```

Both the `myregistry` and global utilities should be available:

```
>>> queryNextUtility(sm1, IMyUtility, 'my_custom_util')
MyUtility('my_custom_util')
>>> queryNextUtility(sm1, IMyUtility, 'myutil')
MyUtility('global')
```

Note, if the context cannot be converted to a site manager, the default is retruned:

```
>>> queryNextUtility(object(), IMyUtility, 'myutil', 'default')
'default'
```

Adapter Registration APIs

Conforming Adapter Lookup

zope.component.**getAdapterInContext**(*object*, *interface*, *context*)

zope.component.**queryAdapterInContext**(*object*, *interface*, *context*, *default=None*)

The *getAdapterInContext()* and *queryAdapterInContext()* APIs first check the context object to see if it already conforms to the requested interface. If so, the object is returned immediately. Otherwise, the adapter factory is looked up in the site manager, and called.

Let's start by creating a component that supports the *__conform__()* method:

```
>>> from zope.interface import implementer
>>> from zope.component.tests.examples import I1
>>> from zope.component.tests.examples import I2
>>> @implementer(I1)
... class Component(object):
...     def __conform__(self, iface, default=None):
...         if iface == I2:
...             return 42
>>> ob = Component()
```

We also gave the component a custom representation, so it will be easier to use in these tests.

We now have to create a site manager (other than the default global one) with which we can register adapters for *I1*.

```
>>> from zope.component.globalregistry import BaseGlobalComponents
>>> sitemanager = BaseGlobalComponents()
```

Now we create a new *context* that knows how to get to our custom site manager.

```
>>> from zope.component.tests.examples import ConformsToIComponentLookup
>>> context = ConformsToIComponentLookup(sitemanager)
```

If an object implements the interface you want to adapt to, *getAdapterInContext()* should simply return the object.

```
>>> from zope.component import getAdapterInContext
>>> from zope.component import queryAdapterInContext
>>> getAdapterInContext(ob, I1, context) is ob
True
>>> queryAdapterInContext(ob, I1, context) is ob
True
```

If an object conforms to the interface you want to adapt to, *getAdapterInContext()* should simply return the conformed object.

```
>>> getAdapterInContext(ob, I2, context)
42
>>> queryAdapterInContext(ob, I2, context)
42
```

If an adapter isn't registered for the given object and interface, and you provide no default, the *getAdapterInContext* API raises ComponentLookupError:

```
>>> from zope.interface import Interface
>>> class I4(Interface):
...     pass

>>> getAdapterInContext(ob, I4, context)
Traceback (most recent call last):
...
ComponentLookupError: (<Component implementing 'I1'>,
                       <InterfaceClass ...I4>)
```

While the *queryAdapterInContext* API returns the default:

```
>>> queryAdapterInContext(ob, I4, context, 44)
44
```

If you ask for an adapter for which something's registered you get the registered adapter:

```
>>> from zope.component.tests.examples import I3
>>> sitemanager.registerAdapter(lambda x: 43, (I1,), I3, '')
>>> getAdapterInContext(ob, I3, context)
43
>>> queryAdapterInContext(ob, I3, context)
43
```

Named Adapter Lookup

zope.component.**getAdapter**(*object*, *interface=<InterfaceClass zope.interface.Interface>*, *name=u''*, *context=None*)

zope.component.**queryAdapter**(*object*, *interface=<InterfaceClass zope.interface.Interface>*, *name=u''*, *default=None*, *context=None*)

The `getAdapter` and `queryAdapter` API functions are similar to `{get|query}AdapterInContext()` functions, except that they do not care about the `__conform__()` but also handle named adapters. (Actually, the name is a required argument.)

If no adapter is registered for the given object, interface, and name, `getAdapter` raises `ComponentLookupError`, while `queryAdapter` returns the default:

```
>>> from zope.component import getAdapter
>>> from zope.component import queryAdapter
>>> from zope.component.tests.examples import I2
>>> from zope.component.tests.examples import ob
>>> getAdapter(ob, I2, '')
Traceback (most recent call last):
...
ComponentLookupError: (<instance Ob>,
```

```
                      <InterfaceClass zope.component.tests.examples.I2>,
                      '')
>>> queryAdapter(ob, I2, '', '<default>')
'<default>'
```

The 'requires' argument to *registerAdapter* must be a sequence, rather than a single interface:

```
>>> from zope.component import getGlobalSiteManager
>>> from zope.component.tests.examples import Comp
>>> gsm = getGlobalSiteManager()
>>> gsm.registerAdapter(Comp, I1, I2, '')
Traceback (most recent call last):
  ...
TypeError: the required argument should be a list of interfaces, not a single␣
↪interface
```

After register an adapter from *I1* to *I2* with the global site manager:

```
>>> from zope.component import getGlobalSiteManager
>>> from zope.component.tests.examples import Comp
>>> gsm = getGlobalSiteManager()
>>> gsm.registerAdapter(Comp, (I1,), I2, '')
```

We can access the adapter using the *getAdapter()* API:

```
>>> adapted = getAdapter(ob, I2, '')
>>> adapted.__class__ is Comp
True
>>> adapted.context is ob
True
>>> adapted = queryAdapter(ob, I2, '')
>>> adapted.__class__ is Comp
True
>>> adapted.context is ob
True
```

If we search using a non-anonymous name, before registering:

```
>>> getAdapter(ob, I2, 'named')
Traceback (most recent call last):
...
ComponentLookupError: (<instance Ob>,
                      <InterfaceClass ....I2>,
                      'named')
>>> queryAdapter(ob, I2, 'named', '<default>')
'<default>'
```

After registering under that name:

```
>>> gsm.registerAdapter(Comp, (I1,), I2, 'named')
>>> adapted = getAdapter(ob, I2, 'named')
>>> adapted.__class__ is Comp
True
>>> adapted.context is ob
True
>>> adapted = queryAdapter(ob, I2, 'named')
>>> adapted.__class__ is Comp
True
```

```
>>> adapted.context is ob
True
```

Invoking an Interface to Perform Adapter Lookup

`zope.component` registers an adapter hook with `zope.interface.interface`, allowing a convenient spelling for adapter lookup: just "call" the interface, passing the context:

```
>>> adapted = I2(ob)
>>> adapted.__class__ is Comp
True
>>> adapted.context is ob
True
```

If the lookup fails, we get a *TypeError*:

```
>>> I2(object())
Traceback (most recent call last):
...
TypeError: ('Could not adapt'...
```

unless we pass a default:

```
>>> marker = object()
>>> adapted = I2(object(), marker)
>>> adapted is marker
True
```

Registering Adapters For Arbitrary Objects

Providing an adapter for None says that your adapter can adapt anything to *I2*.

```
>>> gsm.registerAdapter(Comp, (None,), I2, '')
>>> adapter = I2(ob)
>>> adapter.__class__ is Comp
True
>>> adapter.context is ob
True
```

It can really adapt any arbitrary object:

```
>>> something = object()
>>> adapter = I2(something)
>>> adapter.__class__ is Comp
True
>>> adapter.context is something
True
```

Looking Up Adapters Using Multiple Objects

`zope.component.`**`getMultiAdapter`**(*objects*, *interface=<InterfaceClass zope.interface.Interface>*, *name=u''*, *context=None*)

zope.component.**queryMultiAdapter**(*objects*, *interface=<InterfaceClass zope.interface.Interface>*, *name=u''*, *default=None*, *context=None*)

Multi-adapters adapt one or more objects to another interface. To make this demonstration non-trivial, we need to create a second object to be adapted:

```
>>> from zope.component.tests.examples import Ob2
>>> ob2 = Ob2()
```

As with regular adapters, if an adapter isn't registered for the given objects and interface, the *getMultiAdapter()* API raises *ComponentLookupError*:

```
>>> from zope.component import getMultiAdapter
>>> getMultiAdapter((ob, ob2), I3)
Traceback (most recent call last):
...
ComponentLookupError:
((<instance Ob>, <instance Ob2>),
 <InterfaceClass zope.component.tests.examples.I3>,
 u'')
```

while the *queryMultiAdapter()* API returns the default:

```
>>> from zope.component import queryMultiAdapter
>>> queryMultiAdapter((ob, ob2), I3, default='<default>')
'<default>'
```

Note that `name` is not a required attribute here.

To test multi-adapters, we also have to create an adapter class that handles to context objects:

```
>>> from zope.interface import implementer
>>> @implementer(I3)
... class DoubleAdapter(object):
...     def __init__(self, first, second):
...         self.first = first
...         self.second = second
```

Now we can register the multi-adapter:

```
>>> from zope.component import getGlobalSiteManager
>>> getGlobalSiteManager().registerAdapter(DoubleAdapter, (I1, I2), I3, '')
```

Notice how the required interfaces are simply provided by a tuple.

Now we can get the adapter:

```
>>> adapter = getMultiAdapter((ob, ob2), I3)
>>> adapter.__class__ is DoubleAdapter
True
>>> adapter.first is ob
True
>>> adapter.second is ob2
True
```

Finding More Than One Adapter

zope.component.**getAdapters**(*objects*, *provided*, *context=None*)

It is sometimes desireable to get a list of all adapters that are registered for a particular output interface, given a set of objects.

Let's register some adapters first:

```
>>> class I5(I1):
...     pass
>>> gsm.registerAdapter(Comp, [I1], I5, '')
>>> gsm.registerAdapter(Comp, [None], I5, 'foo')
```

Now we get all the adapters that are registered for `ob` that provide `I5` (note that the names are always text strings, meaning that on Python 2 the names will be `unicode`):

```
>>> from zope.component import getAdapters
>>> adapters = sorted(getAdapters((ob,), I5))
>>> [(str(name), adapter.__class__.__name__) for name, adapter in adapters]
[('', 'Comp'), ('foo', 'Comp')]
>>> try:
...     text = unicode
... except NameError:
...     text = str # Python 3
>>> [isinstance(name, text) for name, _ in adapters]
[True, True]
```

Note that the output doesn't include None values. If an adapter factory returns None, it is as if it wasn't present.

```
>>> gsm.registerAdapter(lambda context: None, [I1], I5, 'nah')
>>> adapters = sorted(getAdapters((ob,), I5))
>>> [(str(name), adapter.__class__.__name__) for name, adapter in adapters]
[('', 'Comp'), ('foo', 'Comp')]
```

Subscription Adapters

`zope.component.`**`subscribers`**(*objects*, *interface*, *context=None*)

Event handlers

`zope.component.`**`handle`**(**objects*)

Helpers for Declaring / Testing Adapters

`zope.component.`**`adapter`**(**interfaces*)

`zope.component.`**`adaptedBy`**(*ob*)

`zope.component.`**`adapts`**(**interfaces*)

Factory APIs

`zope.component.`**`createObject`**(*__factory_name*, **args*, ***kwargs*)

Invoke the named factory and return the result.

`__factory_name` is a positional-only argument.

zope.component.**getFactoryInterfaces**(*name*, *context=None*)
Return the interface provided by the named factory's objects

Result might be a single interface. XXX

zope.component.**getFactoriesFor**(*interface*, *context=None*)
Return info on all factories implementing the given interface.

Interface Registration APIs

Registering Interfaces as Utilities

zope.component.interface.**provideInterface**(*id*, *interface*, *iface_type=None*, *info=''*)
Mark 'interface' as a named utilty providing 'iface_type'.

We can register a given interface with the global site manager as a utility. First, declare a new interface, which itself provides only the core API, `zope.interface.interfaces.IInterface`:

```
>>> from zope.interface import Interface
>>> from zope.interface.interfaces import IInterface
>>> from zope.component.tests.examples import ITestType
>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()

>>> class IDemo(Interface):
...     pass
>>> IInterface.providedBy(IDemo)
True
>>> ITestType.providedBy(IDemo)
False
>>> list(gsm.getUtilitiesFor(ITestType))
[]
```

Now, register `IDemo` as providing `ITestType`

```
>>> from zope.component.interface import provideInterface
>>> provideInterface('', IDemo, ITestType)
>>> ITestType.providedBy(IDemo)
True
>>> interfaces = list(gsm.getUtilitiesFor(ITestType))
>>> [iface.__name__ for (name, iface) in interfaces]
['IDemo']
```

We can register `IDemo` as providing more than one interface:

```
>>> class IOtherType(IInterface):
...     pass
>>> provideInterface('', IDemo, IOtherType)
>>> ITestType.providedBy(IDemo)
True
>>> IOtherType.providedBy(IDemo)
True
>>> interfaces = list(gsm.getUtilitiesFor(ITestType))
>>> [iface.__name__ for (name, iface) in interfaces]
['IDemo']
>>> interfaces = list(gsm.getUtilitiesFor(IOtherType))
```

```
>>> [iface.__name__ for (name, iface) in interfaces]
['IDemo']
```

zope.component.interface.**getInterface**(*context*, *id*)

Return interface or raise ComponentLookupError

```
>>> from zope.interface import Interface
>>> from zope.component.interface import getInterface
>>> from zope.component.tests.examples import ITestType
>>> from zope.component.tests.examples import IGI

>>> IInterface.providedBy(IGI)
True
>>> ITestType.providedBy(IGI)
False
>>> getInterface(None, 'zope.component.tests.examples.IGI')
Traceback (most recent call last):
...
ComponentLookupError: zope.component.tests.examples.interface.IGI
>>> provideInterface('', IGI, ITestType)
>>> ITestType.providedBy(IGI)
True
>>> iface = getInterface(None,
...                      'zope.component.tests.examples.IGI')
>>> iface.__name__
'IGI'
```

zope.component.interface.**queryInterface**(*id*, *default=None*)

Return an interface or None

```
>>> from zope.interface import Interface
>>> from zope.interface.interfaces import IInterface
>>> from zope.component.interface import queryInterface
>>> from zope.component.tests.examples import ITestType
>>> from zope.component.tests.examples import IQI

>>> IInterface.providedBy(IQI)
True
>>> ITestType.providedBy(IQI)
False
>>> queryInterface('zope.component.tests.examples.IQI') is None
True

>>> provideInterface('', IQI, ITestType)
>>> ITestType.providedBy(IQI)
True
>>> iface = queryInterface('zope.component.tests.examples.IQI')
>>> iface.__name__
'IQI'
```

zope.component.interface.**searchInterface**(*context*, *search_string=None*, *base=None*)

Interfaces search

```
>>> from zope.interface import Interface
>>> from zope.interface.interfaces import IInterface
>>> from zope.component.interface import searchInterface
>>> from zope.component.tests.examples import ITestType
>>> from zope.component.tests.examples import ISI
```

```
>>> IInterface.providedBy(ISI)
True
>>> ITestType.providedBy(ISI)
False
>>> searchInterface(None, 'zope.component.tests.examples.ISI')
[]
>>> provideInterface('', ISI, ITestType)
>>> ITestType.providedBy(ISI)
True
>>> searchInterface(None, 'zope.component.tests.examples.ISI') == [ISI]
True
```

zope.component.interface.**searchInterfaceIds**(*context*, *search_string=None*, *base=None*)

Interfaces search

```
>>> from zope.interface import Interface
>>> from zope.interface.interfaces import IInterface
>>> from zope.component.interface import searchInterfaceIds
>>> from zope.component.tests.examples import ITestType
>>> from zope.component.tests.examples import ISII

>>> IInterface.providedBy(ISII)
True
>>> ITestType.providedBy(ISII)
False
>>> searchInterfaceIds(None, 'zope.component.tests.examples.ISII')
[]
>>> provideInterface('', ISII, ITestType)
>>> ITestType.providedBy(ISII)
True
>>> [str(x) for x in searchInterfaceIds(None, 'zope.component.tests.examples.ISII')]
['zope.component.tests.examples.ISII']
```

Security APIs

zope.security support for the configuration handlers

zope.component.security.**securityAdapterFactory**(*factory*, *permission*, *locate*, *trusted*)

If a permission is provided when wrapping the adapter, it will be wrapped in a LocatingAdapterFactory.

```
>>> class Factory(object):
...     pass
```

If both locate and trusted are False and a non-public permission is provided, then the factory is wrapped into a LocatingUntrustedAdapterFactory:

```
>>> from zope.component.security import securityAdapterFactory
>>> from zope.security.adapter import LocatingUntrustedAdapterFactory
>>> factory = securityAdapterFactory(Factory, 'zope.AnotherPermission',
...    locate=False, trusted=False)
>>> isinstance(factory, LocatingUntrustedAdapterFactory)
True
```

If a PublicPermission is provided, then the factory is not touched.

```
>>> from zope.component.security import PublicPermission
>>> factory = securityAdapterFactory(Factory, PublicPermission,
...     locate=False, trusted=False)
>>> factory is Factory
True
```

Same for CheckerPublic:

```
>>> from zope.security.checker import CheckerPublic
>>> factory = securityAdapterFactory(Factory, CheckerPublic,
...     locate=False, trusted=False)
>>> factory is Factory
True
```

If the permission is None, the factory isn't touched:

```
>>> factory = securityAdapterFactory(Factory, None,
...     locate=False, trusted=False)
>>> factory is Factory
True
```

If the factory is trusted and a no permission is provided then the adapter is wrapped into a TrustedAdapterFactory:

```
>>> from zope.security.adapter import TrustedAdapterFactory
>>> factory = securityAdapterFactory(Factory, None,
...     locate=False, trusted=True)
>>> isinstance(factory, TrustedAdapterFactory)
True
```

Same for PublicPermission:

```
>>> factory = securityAdapterFactory(Factory, PublicPermission,
...     locate=False, trusted=True)
>>> isinstance(factory, TrustedAdapterFactory)
True
```

Same for CheckerPublic:

```
>>> factory = securityAdapterFactory(Factory, CheckerPublic,
...     locate=False, trusted=True)
>>> isinstance(factory, TrustedAdapterFactory)
True
```

If the factory is trusted and a locate is true, then the adapter is wrapped into a LocatingTrustedAdapterFactory:

```
>>> from zope.security.adapter import LocatingTrustedAdapterFactory
>>> factory = securityAdapterFactory(Factory, 'zope.AnotherPermission',
...     locate=True, trusted=True)
>>> isinstance(factory, LocatingTrustedAdapterFactory)
True
```

Persistent Registries

Conforming Adapter Lookup

Here, we'll demonstrate that changes work even when data are stored in a database and when accessed from multiple connections.

Start by setting up a database and creating two transaction managers and database connections to work with.

```
>>> import ZODB.tests.util
>>> db = ZODB.tests.util.DB()
>>> import transaction
>>> t1 = transaction.TransactionManager()
>>> c1 = db.open(transaction_manager=t1)
>>> r1 = c1.root()
>>> t2 = transaction.TransactionManager()
>>> c2 = db.open(transaction_manager=t2)
>>> r2 = c2.root()
```

Create a set of components registries in the database, alternating connections.

```
>>> from zope.component.persistentregistry import PersistentComponents
>>> from zope.component.tests.examples import I1
>>> from zope.component.tests.examples import I2
>>> from zope.component.tests.examples import U
>>> from zope.component.tests.examples import U1
>>> from zope.component.tests.examples import U12
>>> from zope.component.tests.examples import handle1
>>> from zope.component.tests.examples import handle2
>>> from zope.component.tests.examples import handle3
>>> from zope.component.tests.examples import handle4

>>> _ = t1.begin()
>>> r1[1] = PersistentComponents('1')
>>> t1.commit()

>>> _ = t2.begin()
>>> r2[2] = PersistentComponents('2', (r2[1], ))
>>> t2.commit()

>>> _ = t1.begin()
>>> r1[3] = PersistentComponents('3', (r1[1], ))
>>> t1.commit()

>>> _ = t2.begin()
>>> r2[4] = PersistentComponents('4', (r2[2], r2[3]))
>>> t2.commit()

>>> _ = t1.begin()
>>> r1[1].__bases__
()
>>> r1[2].__bases__ == (r1[1], )
True

>>> r1[1].registerUtility(U1(1))
>>> r1[1].queryUtility(I1)
U1(1)
>>> r1[2].queryUtility(I1)
```

```
U1(1)
>>> t1.commit()

>>> _ = t2.begin()
>>> r2[1].registerUtility(U1(2))
>>> r2[2].queryUtility(I1)
U1(2)

>>> r2[4].queryUtility(I1)
U1(2)
>>> t2.commit()


>>> _ = t1.begin()
>>> r1[1].registerUtility(U12(1), I2)
>>> r1[4].queryUtility(I2)
U12(1)
>>> t1.commit()


>>> _ = t2.begin()
>>> r2[3].registerUtility(U12(3), I2)
>>> r2[4].queryUtility(I2)
U12(3)
>>> t2.commit()

>>> _ = t1.begin()

>>> r1[1].registerHandler(handle1, info="First handler")
>>> r1[2].registerHandler(handle2, required=[U])

>>> r1[3].registerHandler(handle3)

>>> r1[4].registerHandler(handle4)

>>> r1[4].handle(U1(1))
handle1 U1(1)
handle3 U1(1)
handle2 (U1(1),)
handle4 U1(1)

>>> t1.commit()

>>> _ = t2.begin()
>>> r2[4].handle(U1(1))
handle1 U1(1)
handle3 U1(1)
handle2 (U1(1),)
handle4 U1(1)
>>> t2.abort()

>>> db.close()
```

Subscription to Events in Persistent Registries

```
>>> import ZODB.tests.util
>>> db = ZODB.tests.util.DB()
>>> import transaction
>>> t1 = transaction.TransactionManager()
>>> c1 = db.open(transaction_manager=t1)
>>> r1 = c1.root()
>>> t2 = transaction.TransactionManager()
>>> c2 = db.open(transaction_manager=t2)
>>> r2 = c2.root()

>>> from zope.component.persistentregistry import PersistentComponents

>>> _ = t1.begin()
>>> r1[1] = PersistentComponents('1')
>>> r1[1].registerHandler(handle1)
>>> r1[1].registerSubscriptionAdapter(handle1, provided=I2)
>>> _ = r1[1].unregisterHandler(handle1)
>>> _ = r1[1].unregisterSubscriptionAdapter(handle1, provided=I2)
>>> t1.commit()
>>> _ = t1.begin()
>>> r1[1].registerHandler(handle1)
>>> r1[1].registerSubscriptionAdapter(handle1, provided=I2)
>>> t1.commit()

>>> _ = t2.begin()
>>> len(list(r2[1].registeredHandlers()))
1
>>> len(list(r2[1].registeredSubscriptionAdapters()))
1
>>> t2.abort()
```

Adapter Registrations after Serialization / Deserialization

We want to make sure that we see updates correctly.

```
>>> import persistent
>>> import transaction
>>> from zope.interface import Interface
>>> from zope.interface import implementer
>>> class IFoo(Interface):
...     pass
>>> @implementer(IFoo)
... class Foo(persistent.Persistent):
...     name = ''
...     def __init__(self, name=''):
...         self.name = name
...     def __repr__(self):
...         return 'Foo(%r)' % self.name

>>> from zope.component.tests.examples import base
>>> from zope.component.tests.examples import clear_base
>>> len(base._v_subregistries)
0
```

```
>>> import ZODB.tests.util
>>> db = ZODB.tests.util.DB()
>>> tm1 = transaction.TransactionManager()
>>> c1 = db.open(transaction_manager=tm1)
>>> from zope.component.persistentregistry import PersistentAdapterRegistry
>>> r1 = PersistentAdapterRegistry((base,))
>>> r2 = PersistentAdapterRegistry((r1,))
>>> c1.root()[1] = r1
>>> c1.root()[2] = r2
>>> tm1.commit()
>>> r1._p_deactivate()

>>> len(base._v_subregistries)
0

>>> tm2 = transaction.TransactionManager()
>>> c2 = db.open(transaction_manager=tm2)
>>> r1 = c2.root()[1]
>>> r2 = c2.root()[2]

>>> r1.lookup((), IFoo, '')

>>> base.register((), IFoo, '', Foo(''))
>>> r1.lookup((), IFoo, '')
Foo('')

>>> r2.lookup((), IFoo, '1')

>>> r1.register((), IFoo, '1', Foo('1'))

>>> r2.lookup((), IFoo, '1')
Foo('1')

>>> r1.lookup((), IFoo, '2')
>>> r2.lookup((), IFoo, '2')

>>> base.register((), IFoo, '2', Foo('2'))

>>> r1.lookup((), IFoo, '2')
Foo('2')

>>> r2.lookup((), IFoo, '2')
Foo('2')

>>> db.close()
>>> clear_base()
```

CHAPTER 11

Hacking on `zope.component`

Getting the Code

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

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

You can get a read-only checkout from there:

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

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

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

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

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

You can branch the trunk from there using Bazaar:

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

Working in a `virtualenv`

Installing

If you use the `virtualenv` package to create lightweight Python development environments, you can run the tests using nothing more than the `python` binary in a virtualenv. First, create a scratch environment:

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

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

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

Running the tests

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

```
$ /tmp/hack-zope.component/bin/python setup.py test -q
.....................................................................................
 ↪.....................................................................................
 ↪.....................................................................
----------------------------------------------------------------------
Ran 249 tests in 0.000s

OK
```

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

```
$ /tmp/hack-zope.component/bin/nosetests
.....................................................................................
 ↪.....................................................................................
 ↪.....................................................................................
 ↪.........
----------------------------------------------------------------------
Ran 263 tests in 0.000s

OK
```

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

```
$ /tmp/hack-zope.component/bin/easy_install nose coverage
...
$ /tmp/hack-zope.component/bin/nosetests --with coverage
.....................................................................................
 ↪.....................................................................................
 ↪.....................................................................................
 ↪.........
Name                                    Stmts   Miss  Cover   Missing
---------------------------------------------------------------------
zope/component.py                          42      0   100%
zope/component/_api.py                    132      0   100%
zope/component/_compat.py                   3      0   100%
zope/component/_declaration.py             30      0   100%
zope/component/event.py                    10      0   100%
zope/component/eventtesting.py             11      0   100%
zope/component/factory.py                  20      0   100%
zope/component/globalregistry.py           38      0   100%
zope/component/hookable.py                 14      0   100%
zope/component/hooks.py                    70      0   100%
zope/component/interface.py                63      0   100%
zope/component/interfaces.py               63      0   100%
zope/component/persistentregistry.py       32      0   100%
zope/component/registry.py                 24      0   100%
zope/component/security.py                 65      0   100%
zope/component/standalonetests.py           2      0   100%
zope/component/zcml.py                    207      0   100%
---------------------------------------------------------------------
```

```
TOTAL                                      826      0   100%
----------------------------------------------------------------------
Ran 263 tests in 0.000s

OK
```

Building the documentation

`zope.component` 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.component/bin/easy_install \
 Sphinx repoze.sphinx.autoitnerface zope.component
...
$ cd docs
$ PATH=/tmp/hack-zope.component/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.component/bin:$PATH make doctest
sphinx-build -b doctest -d _build/doctrees   . _build/doctest
...
running tests...

...

Doctest summary
===============
  964 tests
    0 failures in tests
    0 failures in setup code
    0 failures in cleanup code
build succeeded.
Testing of doctests in the sources finished, look at the  results in _build/doctest/
↪output.txt.
```

Using `zc.buildout`

Setting up the buildout

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

```
$ /path/to/python2.7 bootstrap.py
...
Generated script '.../bin/buildout'
$ bin/buildout
Develop: '/home/jrandom/projects/Zope/zope.component/.'
```

```
...
Got coverage 3.7.1
```

Running the tests

You can now run the tests:

```
$ bin/test --all
Running zope.testing.testrunner.layer.UnitTests tests:
  Set up zope.testing.testrunner.layer.UnitTests in 0.000 seconds.
  Ran 249 tests with 0 failures and 0 errors in 0.000 seconds.
Tearing down left over layers:
  Tear down zope.testing.testrunner.layer.UnitTests in 0.000 seconds.
```

Using `tox`

Running Tests on Multiple Python Versions

tox is a Python-based test automation tool designed to run tests against multiple Python versions. It creates a `virtualenv` for each configured version, installs the current package and configured dependencies into each `virtualenv`, and then runs the configured commands.

`zope.component` configures the following `tox` environments via its `tox.ini` file:

- The `py26`, `py27`, `py33`, `py34`, and `pypy` environments builds a `virtualenv` with the appropriate interpreter, installs `zope.component` and dependencies, and runs the tests via `python setup.py test -q`.
- The `coverage` environment builds a `virtualenv` with `python2.6`, installs `zope.component`, installs `nose` and `coverage`, and runs `nosetests` with statement coverage.
- The `docs` environment builds a virtualenv with `python2.6`, installs `zope.component`, 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: /home/tseaver/projects/Zope/Z3/zopetoolkit/src/zope.component/setup.
↪py
py26 inst-nodeps: /home/tseaver/projects/Zope/Z3/zopetoolkit/src/zope.component/.tox/
↪dist/zope.component-4.2.2.dev0.zip
py26 runtests: PYTHONHASHSEED='3711600167'
py26 runtests: commands[0] | python setup.py test -q
running test

...

running build_ext
.....................................................................................
↪.....................................................................................
↪.....................................
----------------------------------------------------------------------
Ran 249 tests 0.000s

OK
_____________________________________ summary _____________________________________
```

```
  py26: commands succeeded
  congratulations :)
```

Running `tox` with no arguments runs all the configured environments, including building the docs and testing their snippets:

```
$ tox
GLOB sdist-make: .../zope.component/setup.py
py26 sdist-reinst: .../zope.component/.tox/dist/zope.component-4.0.2dev.zip
...
Doctest summary
===============
  964 tests
    0 failures in tests
    0 failures in setup code
    0 failures in cleanup code
build succeeded.
_____________________________________ summary ____________________________________
  py26: commands succeeded
  py26min: commands succeeded
  py27: commands succeeded
  pypy: commands succeeded
  py32: commands succeeded
  py33: commands succeeded
  py34: commands succeeded
  coverage: commands succeeded
  docs: commands succeeded
  congratulations :)
```

Contributing to `zope.component`

Submitting a Bug Report

`zope.component` tracks its bugs on Github:

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

Please submit bug reports and feature requests there.

Sharing Your Changes

Note: Please ensure that all tests are passing before you submit your code. If possible, your submission should include new tests for new features or bug fixes, although it is possible that you may have tested your new code by updating existing tests.

If have made a change you would like to share, the best route is to fork the Githb repository, check out your fork, make your changes on a branch in your fork, and push it. You can then submit a pull request from your branch:

https://github.com/zopefoundation/zope.component/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.component/cool_feature
```

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

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

Python Module Index

z

Index

Symbols

A

C

D

G

H

I

M

P

Q

R

S

T

Z