Demo Demo

Component-Management objects

Component-management objects provide a higher-level component-management API over the basic adapter-registration API provided by the zope.interface package. In particular, it provides:

- utilities
- support for computing adapters, rather than just looking up adapter factories.
- management of registration comments

The zope.component.registry.Components class provides an implementation of zope.component.interfaces.IComponents that provides these features.

```
>>> from zope.component import registry
>>> from zope.component import tests
>>> components = registry.Components('comps')
```

As components are registered, events are generated. Let's register an event subscriber, so we can see the events generated:

```
>>> import zope.event
>>> def logevent(event):
...     print event
>>> zope.event.subscribers.append(logevent)
```

Utilities

You can register Utilities using registerUtility:

```
>>> components.registerUtility(tests.U1(1))
Registered event:
UtilityRegistration(<Components comps>, I1, u'', 1, u'')
```

Here we didn't specify an interface or name. An unnamed utility was registered for interface I1, since that is only interface implemented by the U1 class:

```
>>> components.getUtility(tests.I1)
U1(1)
```

If a component implements other than one interface or no interface, then an error will be raised:

```
>>> components.registerUtility(tests.U12(2))
... # doctest: +NORMALIZE_WHITESPACE
Traceback (most recent call last):
...
TypeError: The utility doesn't provide a single interface and
no provided interface was specified.
>>> components.registerUtility(tests.A)
... # doctest: +NORMALIZE_WHITESPACE
Traceback (most recent call last):
...
TypeError: The utility doesn't provide a single interface and
no provided interface was specified.
```

We can provide an interface if desired:

```
>>> components.registerUtility(tests.U12(2), tests.I2)
Registered event:
UtilityRegistration(<Components comps>, I2, u'', 2, u'')
```

and we can specify a name:

```
>>> components.registerUtility(tests.U12(3), tests.I2, u'three')
Registered event:
UtilityRegistration(<Components comps>, I2, u'three', 3, u'')
>>> components.getUtility(tests.I2)
U12(2)
>>> components.getUtility(tests.I2, 'three')
U12(3)
```

If you try to get a utility that doesn't exist, you'll get a component lookup error:

```
>>> components.getUtility(tests.I3)
... # doctest: +NORMALIZE_WHITESPACE
Traceback (most recent call last):
...
ComponentLookupError:
(<InterfaceClass zope.component.tests.I3>, u'')
```

Unless you use queryUtility:

```
>>> components.queryUtility(tests.I3)
>>> components.queryUtility(tests.I3, default=42)
42
```

You can get information about registered utilities with the registeredUtilities method:

```
>>> for registration in sorted(components.registeredUtilities()):
...     print registration.provided, registration.name
...     print registration.component, registration.info
<InterfaceClass zope.component.tests.I1>
U1(1)
<InterfaceClass zope.component.tests.I2>
U12(2)
<InterfaceClass zope.component.tests.I2> three
U12(3)
```

Duplicate registrations replace existing ones:

```
>>> components.registerUtility(tests.U1(4), info=u'use 4 now')
Registered event:
UtilityRegistration(<Components comps>, I1, u'', 4, u'use 4 now')
>>> components.getUtility(tests.I1)
U1(4)
>>> for registration in sorted(components.registeredUtilities()):
...     print registration.provided, registration.name
...     print registration.component, registration.info
<InterfaceClass zope.component.tests.I1>
U1(4) use 4 now
<InterfaceClass zope.component.tests.I2>
U12(2)
<InterfaceClass zope.component.tests.I2> three
U12(3)
```

As shown in the this example, you can provide an "info" argumemnt when registering utilities. This provides extra documentation about the registration itself that is shown when listing registrations.

You can also unregister utilities:

```
>>> components.unregisterUtility(provided=tests.I1)
Unregistered event:
UtilityRegistration(<Components comps>, I1, u'', 4, u'use 4 now')
True
```

A boolean is returned indicating whether anything changed:

```
>>> components.queryUtility(tests.I1)
>>> for registration in sorted(components.registeredUtilities()):
...     print registration.provided, registration.name
...     print registration.component, registration.info
<InterfaceClass zope.component.tests.I2>
U12(2)
<InterfaceClass zope.component.tests.I2> three
U12(3)
```

When you unregister, you can specify a component. If the component doesn't match the one registered, then nothing happens:

```
>>> u5 = tests.U1(5)
>>> components.registerUtility(u5)
Registered event:
UtilityRegistration(<Components comps>, I1, u'', 5, u'')
>>> components.unregisterUtility(tests.U1(6))
False
>>> components.queryUtility(tests.I1)
U1(5)
>>> components.unregisterUtility(u5)
Unregistered event:
UtilityRegistration(<Components comps>, I1, u'', 5, u'')
True
>>> components.queryUtility(tests.I1)
```

You can get the name and utility for all of the utilities that provide an interface using getUtilitiesFor:

```
>>> sorted(components.getUtilitiesFor(tests.I2))
[(u'', U12(2)), (u'three', U12(3))]
```

getAllUtilitiesRegisteredFor is similar to getUtilitiesFor except that it includes utilities that are overridden. For example, we'll register a utility that for an extending interface of I2:

```
>>> components.registerUtility(tests.U('ext'), tests.I2e)
Registered event:
UtilityRegistration(<Components comps>, I2e, u'', ext, u'')
```

We don't get the new utility for getUtilitiesFor:

```
>>> sorted(components.getUtilitiesFor(tests.I2))
[(u'', U12(2)), (u'three', U12(3))]
```

but we do get it from getAllUtilitiesRegisteredFor:

```
>>> sorted(map(str, components.getAllUtilitiesRegisteredFor(tests.I2)))
['U(ext)', 'U12(2)', 'U12(3)']
```

Adapters

You can register adapters with registerAdapter:

```
>>> components.registerAdapter(tests.A12_1)
Registered event:
AdapterRegistration(<Components comps>, [I1, I2], IA1, u'', A12_1, u'')
```

Here, we didn't specify required interfaces, a provided interface, or a name. The required interfaces were determined from the factory s __component_adapts__ attribute and the provided interface was determined by introspecting what the factory implements.

```
>>> components.getMultiAdapter((tests.U1(6), tests.U12(7)), tests.IA1)
A12_1(U1(6), U12(7))
```

If a factory implements more than one interface, an exception will be raised:

```
>>> components.registerAdapter(tests.A1_12)
... # doctest: +NORMALIZE_WHITESPACE
Traceback (most recent call last):
...
TypeError: The adapter factory doesn't implement a single
interface and no provided interface was specified.
```

Unless the provided interface is specified:

```
>>> components.registerAdapter(tests.A1_12, provided=tests.IA2)
Registered event:
AdapterRegistration(<Components comps>, [I1], IA2, u'', A1_12, u'')
```

If a factory doesn't declare an implemented interface, an exception will be raised:

```
>>> components.registerAdapter(tests.A12_)
... # doctest: +NORMALIZE_WHITESPACE
Traceback (most recent call last):
...
TypeError: The adapter factory doesn't implement a single
interface and no provided interface was specified.
```

Unless the provided interface is specified:

```
>>> components.registerAdapter(tests.A12_, provided=tests.IA2)
Registered event:
AdapterRegistration(<Components comps>, [I1, I2], IA2, u'', A12_, u'')
```

The required interface needs to be specified in the registration if the factory doesn't have a __component_adapts__ attribute:

```
>>> components.registerAdapter(tests.A_2)
... # doctest: +NORMALIZE_WHITESPACE
Traceback (most recent call last):
...
TypeError: The adapter factory doesn't have a __component_adapts__
attribute and no required specifications were specified
```

Unless the required specifications specified:

```
>>> components.registerAdapter(tests.A_2, required=[tests.I3])
Registered event:
AdapterRegistration(<Components comps>, [I3], IA2, u'', A_2, u'')
```

Classes can be specified in place of specifications, in which case the implementedBy specification for the class is used:

```
>>> components.registerAdapter(tests.A_3, required=[tests.U],
...                            info="Really class specific")
... # doctest: +NORMALIZE_WHITESPACE
Registered event:
AdapterRegistration(<Components comps>, [zope.component.tests.U], IA3, u'',
                    A_3, 'Really class specific')
```

We can see the adapters that have been registered using the registeredAdapters method:

```
>>> for registration in sorted(components.registeredAdapters()):
...     print registration.required
...     print registration.provided, registration.name
...     print registration.factory, registration.info
... # doctest: +NORMALIZE_WHITESPACE
(<InterfaceClass zope.component.tests.I1>,
 <InterfaceClass zope.component.tests.I2>)
<InterfaceClass zope.component.tests.IA1>
zope.component.tests.A12_1
(<InterfaceClass zope.component.tests.I1>,
 <InterfaceClass zope.component.tests.I2>)
<InterfaceClass zope.component.tests.IA2>
zope.component.tests.A12_
(<InterfaceClass zope.component.tests.I1>,)
<InterfaceClass zope.component.tests.IA2>
zope.component.tests.A1_12
(<InterfaceClass zope.component.tests.I3>,)
<InterfaceClass zope.component.tests.IA2>
zope.component.tests.A_2
(<implementedBy zope.component.tests.U>,)
<InterfaceClass zope.component.tests.IA3>
zope.component.tests.A_3 Really class specific
```

As with utilities, we can provide registration information when registering adapters.

If you try to fetch an adapter that isn't registered, you'll get a component-lookup error:

```
>>> components.getMultiAdapter((tests.U(8), ), tests.IA1)
... # doctest: +NORMALIZE_WHITESPACE
Traceback (most recent call last):
...
ComponentLookupError: ((U(8),),
                       <InterfaceClass zope.component.tests.IA1>, u'')
```

unless you use queryAdapter:

```
>>> components.queryMultiAdapter((tests.U(8), ), tests.IA1)
>>> components.queryMultiAdapter((tests.U(8), ), tests.IA1, default=42)
42
```

When looking up an adapter for a single object, you can use the slightly simpler getAdapter and queryAdapter calls:

```
>>> components.getAdapter(tests.U1(9), tests.IA2)
A1_12(U1(9))
```

```
>>> components.queryAdapter(tests.U1(9), tests.IA2)
A1_12(U1(9))
>>> components.getAdapter(tests.U(8), tests.IA1)
... # doctest: +NORMALIZE_WHITESPACE
Traceback (most recent call last):
...
ComponentLookupError: (U(8),
                       <InterfaceClass zope.component.tests.IA1>, u'')
>>> components.queryAdapter(tests.U(8), tests.IA2)
>>> components.queryAdapter(tests.U(8), tests.IA2, default=42)
42
```

You can unregister an adapter. If a factory is provided and if the rewuired and provided interfaces, can be infered, then they need not be provided:

```
>>> components.unregisterAdapter(tests.A12_1)
Unregistered event:
AdapterRegistration(<Components comps>, [I1, I2], IA1, u'', A12_1, u'')
True
>>> for registration in sorted(components.registeredAdapters()):
...     print registration.required
...     print registration.provided, registration.name
...     print registration.factory, registration.info
... # doctest: +NORMALIZE_WHITESPACE
(<InterfaceClass zope.component.tests.I1>,
 <InterfaceClass zope.component.tests.I2>)
<InterfaceClass zope.component.tests.IA2>
zope.component.tests.A12_
(<InterfaceClass zope.component.tests.I1>,)
<InterfaceClass zope.component.tests.IA2>
zope.component.tests.A1_12
(<InterfaceClass zope.component.tests.I3>,)
<InterfaceClass zope.component.tests.IA2>
zope.component.tests.A_2
(<implementedBy zope.component.tests.U>,)
<InterfaceClass zope.component.tests.IA3>
zope.component.tests.A_3 Really class specific
```

A boolean is returned indicating whether a change was made.

If a factory implements more than one interface, an exception will be raised:

```
>>> components.unregisterAdapter(tests.A1_12)
... # doctest: +NORMALIZE_WHITESPACE
Traceback (most recent call last):
...
TypeError: The adapter factory doesn't implement a single
interface and no provided interface was specified.
```

Unless the provided interface is specified:

```
>>> components.unregisterAdapter(tests.A1_12, provided=tests.IA2)
Unregistered event:
AdapterRegistration(<Components comps>, [I1], IA2, u'', A1_12, u'')
True
```

If a factory doesn't declare an implemented interface, an exception will be raised:

```
>>> components.unregisterAdapter(tests.A12_)
... # doctest: +NORMALIZE_WHITESPACE
```

```
Traceback (most recent call last):
...
TypeError: The adapter factory doesn't implement a single
interface and no provided interface was specified.
```

Unless the provided interface is specified:

```
>>> components.unregisterAdapter(tests.A12_, provided=tests.IA2)
Unregistered event:
AdapterRegistration(<Components comps>, [I1, I2], IA2, u'', A12_, u'')
True
```

The required interface needs to be specified if the factory doesn't have a __component_adapts__ attribute:

```
>>> components.unregisterAdapter(tests.A_2)
... # doctest: +NORMALIZE_WHITESPACE
Traceback (most recent call last):
...
TypeError: The adapter factory doesn't have a __component_adapts__
attribute and no required specifications were specified
>>> components.unregisterAdapter(tests.A_2, required=[tests.I3])
Unregistered event:
AdapterRegistration(<Components comps>, [I3], IA2, u'', A_2, u'')
True
>>> for registration in sorted(components.registeredAdapters()):
...     print registration.required
...     print registration.provided, registration.name
...     print registration.factory, registration.info
... # doctest: +NORMALIZE_WHITESPACE
(<implementedBy zope.component.tests.U>,)
<InterfaceClass zope.component.tests.IA3>
zope.component.tests.A_3 Really class specific
```

If a factory is unregistered that is not registered, False is returned:

```
>>> components.unregisterAdapter(tests.A_2, required=[tests.I3])
False
>>> components.unregisterAdapter(tests.A12_1, required=[tests.U])
False
```

The factory can be omitted, to unregister *any* factory that matches specified required and provided interfaces:

```
>>> components.unregisterAdapter(required=[tests.U], provided=tests.IA3)
... # doctest: +NORMALIZE_WHITESPACE
Unregistered event:
AdapterRegistration(<Components comps>, [zope.component.tests.U],
                    IA3, u'', A_3, 'Really class specific')
True
>>> for registration in sorted(components.registeredAdapters()):
...     print registration
```

Adapters can be named:

```
>>> components.registerAdapter(tests.A1_12, provided=tests.IA2,
...                            name=u'test')
Registered event:
AdapterRegistration(<Components comps>, [I1], IA2, u'test', A1_12, u'')
>>> components.queryMultiAdapter((tests.U1(9), ), tests.IA2)
>>> components.queryMultiAdapter((tests.U1(9), ), tests.IA2, name=u'test')
A1_12(U1(9))
```

```
>>> components.queryAdapter(tests.U1(9), tests.IA2)
>>> components.queryAdapter(tests.U1(9), tests.IA2, name=u'test')
A1_12(U1(9))
>>> components.getAdapter(tests.U1(9), tests.IA2, name=u'test')
A1_12(U1(9))
```

It is possible to look up all of the adapters that provide an interface:

```
>>> components.registerAdapter(tests.A1_23, provided=tests.IA2,
...                            name=u'test 2')
Registered event:
AdapterRegistration(<Components comps>, [I1], IA2, u'test 2', A1_23, u'')
>>> components.registerAdapter(tests.A1_12, provided=tests.IA2)
Registered event:
AdapterRegistration(<Components comps>, [I1], IA2, u'', A1_12, u'')
>>> for name, adapter in sorted(components.getAdapters((tests.U1(9), ),
...                                                    tests.IA2)):
...     print name, adapter
 A1_12(U1(9))
test A1_12(U1(9))
test 2 A1_23(U1(9))
```

getAdapters is most commonly used as the basis of menu systems.

If an adapter factory returns None, it is equivalent to there being no factory:

```
>>> components.registerAdapter(tests.noop,
...                            required=[tests.IA1], provided=tests.IA2,
...                            name=u'test noop')
... # doctest: +NORMALIZE_WHITESPACE
Registered event:
AdapterRegistration(<Components comps>, [IA1], IA2, u'test noop',
                    noop, u'')
>>> components.queryAdapter(tests.U1(9), tests.IA2, name=u'test noop')
>>> components.registerAdapter(tests.A1_12, provided=tests.IA2)
Registered event:
AdapterRegistration(<Components comps>, [I1], IA2, u'', A1_12, u'')
>>> for name, adapter in sorted(components.getAdapters((tests.U1(9), ),
...                                                    tests.IA2)):
...     print name, adapter
 A1_12(U1(9))
test A1_12(U1(9))
test 2 A1_23(U1(9))
>>> components.unregisterAdapter(tests.A1_12, provided=tests.IA2,
...                              name=u'test')
Unregistered event:
AdapterRegistration(<Components comps>, [I1], IA2, u'test', A1_12, u'')
True
>>> components.unregisterAdapter(tests.A1_12, provided=tests.IA2)
Unregistered event:
AdapterRegistration(<Components comps>, [I1], IA2, u'', A1_12, u'')
True
>>> for registration in sorted(components.registeredAdapters()):
...     print registration.required
...     print registration.provided, registration.name
...     print registration.factory, registration.info
... # doctest: +NORMALIZE_WHITESPACE
(<InterfaceClass zope.component.tests.I1>,)
<InterfaceClass zope.component.tests.IA2> test 2
zope.component.tests.A1_23
```

```
(<InterfaceClass zope.component.tests.IA1>,)
<InterfaceClass zope.component.tests.IA2> test noop
<function noop at 0xb79a1064>
```

Subscribers

Subscribers provide a way to get multiple adapters of a given type. In this regard, subscribers are like named adapters, except that there isn't any concept of the most specific adapter for a given name.

Subscribers are registered by calling registerSubscriptionAdapter:

```
>>> components.registerSubscriptionAdapter(tests.A1_2)
... # doctest: +NORMALIZE_WHITESPACE
Registered event:
SubscriptionRegistration(<Components comps>, [I1], IA2, u'', A1_2, u'')
>>> components.registerSubscriptionAdapter(
...     tests.A1_12, provided=tests.IA2)
... # doctest: +NORMALIZE_WHITESPACE
Registered event:
SubscriptionRegistration(<Components comps>, [I1], IA2, u'', A1_12, u'')
>>> components.registerSubscriptionAdapter(
...     tests.A, [tests.I1], tests.IA2,
...     info='a sample comment')
... # doctest: +NORMALIZE_WHITESPACE
Registered event:
SubscriptionRegistration(<Components comps>, [I1], IA2, u'',
                         A, 'a sample comment')
```

The same rules, with regard to when required and provided interfaces have to be specified apply as with adapters:

```
>>> components.registerSubscriptionAdapter(tests.A1_12)
... # doctest: +NORMALIZE_WHITESPACE
Traceback (most recent call last):
...
TypeError: The adapter factory doesn't implement a single
interface and no provided interface was specified.
>>> components.registerSubscriptionAdapter(tests.A)
... # doctest: +NORMALIZE_WHITESPACE
Traceback (most recent call last):
...
TypeError: The adapter factory doesn't implement a single interface and
 no provided interface was specified.
>>> components.registerSubscriptionAdapter(tests.A, required=[tests.IA1])
... # doctest: +NORMALIZE_WHITESPACE
Traceback (most recent call last):
...
TypeError: The adapter factory doesn't implement a single interface
and no provided interface was specified.
```

Note that we provided the info argument as a keyword argument above. That's because there is a name argument that's reserved for future use. We can give a name, as long as it is an empty string:

```
>>> components.registerSubscriptionAdapter(
...     tests.A, [tests.I1], tests.IA2, u'', 'a sample comment')
... # doctest: +NORMALIZE_WHITESPACE
Registered event:
SubscriptionRegistration(<Components comps>, [I1], IA2, u'',
                         A, 'a sample comment')
```

```
>>> components.registerSubscriptionAdapter(
...     tests.A, [tests.I1], tests.IA2, u'oops', 'a sample comment')
Traceback (most recent call last):
...
TypeError: Named subscribers are not yet supported
```

Subscribers are looked up using the subscribers method:

```
>>> for s in components.subscribers((tests.U1(1), ), tests.IA2):
...    print s
A1_2(U1(1))
A1_12(U1(1))
A(U1(1),)
A(U1(1),)
```

Note that, because we created multiple subscriptions for A, we got multiple subscriber instances.

As with normal adapters, if a factory returns None, the result is skipped:

```
>>> components.registerSubscriptionAdapter(
...     tests.noop, [tests.I1], tests.IA2)
Registered event:
SubscriptionRegistration(<Components comps>, [I1], IA2, u'', noop, u'')
>>> for s in components.subscribers((tests.U1(1), ), tests.IA2):
...    print s
A1_2(U1(1))
A1_12(U1(1))
A(U1(1),)
A(U1(1),)
```

We can get registration information for subscriptions:

```
>>> for registration in sorted(
...     components.registeredSubscriptionAdapters()):
...     print registration.required
...     print registration.provided, registration.name
...     print registration.factory, registration.info
(<InterfaceClass zope.component.tests.I1>,)
<InterfaceClass zope.component.tests.IA2>
zope.component.tests.A a sample comment
(<InterfaceClass zope.component.tests.I1>,)
<InterfaceClass zope.component.tests.IA2>
zope.component.tests.A a sample comment
(<InterfaceClass zope.component.tests.I1>,)
<InterfaceClass zope.component.tests.IA2>
zope.component.tests.A1_12
(<InterfaceClass zope.component.tests.I1>,)
<InterfaceClass zope.component.tests.IA2>
zope.component.tests.A1_2
(<InterfaceClass zope.component.tests.I1>,)
<InterfaceClass zope.component.tests.IA2>
<function noop at 0xb796ff7c>
```

We can also unregister subscriptions in much the same way we can for adapters:

```
>>> components.unregisterSubscriptionAdapter(tests.A1_2)
... # doctest: +NORMALIZE_WHITESPACE
Unregistered event:
SubscriptionRegistration(<Components comps>, [I1], IA2, u'', A1_2, '')
True
```

```
>>> for s in components.subscribers((tests.U1(1), ), tests.IA2):
...    print s
A1_12(U1(1))
A(U1(1),)
A(U1(1),)
>>> for registration in sorted(
...     components.registeredSubscriptionAdapters()):
...     print registration.required
...     print registration.provided, registration.name
...     print registration.factory, registration.info
(<InterfaceClass zope.component.tests.I1>,)
<InterfaceClass zope.component.tests.IA2>
zope.component.tests.A a sample comment
(<InterfaceClass zope.component.tests.I1>,)
<InterfaceClass zope.component.tests.IA2>
zope.component.tests.A a sample comment
(<InterfaceClass zope.component.tests.I1>,)
<InterfaceClass zope.component.tests.IA2>
zope.component.tests.A1_12
(<InterfaceClass zope.component.tests.I1>,)
<InterfaceClass zope.component.tests.IA2>
<function noop at 0xb796ff7c>
>>> components.unregisterSubscriptionAdapter(
...     tests.A, [tests.I1], tests.IA2)
Unregistered event:
SubscriptionRegistration(<Components comps>, [I1], IA2, u'', A, '')
True
>>> for s in components.subscribers((tests.U1(1), ), tests.IA2):
...    print s
A1_12(U1(1))
>>> for registration in sorted(
...     components.registeredSubscriptionAdapters()):
...     print registration.required
...     print registration.provided, registration.name
...     print registration.factory, registration.info
(<InterfaceClass zope.component.tests.I1>,)
<InterfaceClass zope.component.tests.IA2>
zope.component.tests.A1_12
(<InterfaceClass zope.component.tests.I1>,)
<InterfaceClass zope.component.tests.IA2>
<function noop at 0xb796ff7c>
```

Note here that both registrations for A were removed.

If we omit the factory, we must specify the required and provided interfaces:

```
>>> components.unregisterSubscriptionAdapter(required=[tests.I1])
Traceback (most recent call last):
...
TypeError: Must specify one of factory and provided
>>> components.unregisterSubscriptionAdapter(provided=tests.IA2)
Traceback (most recent call last):
...
TypeError: Must specify one of factory and required
>>> components.unregisterSubscriptionAdapter(
...     required=[tests.I1], provided=tests.IA2)
Unregistered event:
SubscriptionRegistration(<Components comps>, [I1], IA2, u'', None, '')
True
```

```
>>> for s in components.subscribers((tests.U1(1), ), tests.IA2):
...     print s
>>> for registration in sorted(
...      components.registeredSubscriptionAdapters()):
...      print registration.factory
```

As when registering, an error is raised if the registration information can't be determined from the factory and isn't specified:

```
>>> components.unregisterSubscriptionAdapter(tests.A1_12)
... # doctest: +NORMALIZE_WHITESPACE
Traceback (most recent call last):
...
TypeError: The adapter factory doesn't implement a single
interface and no provided interface was specified.
>>> components.unregisterSubscriptionAdapter(tests.A)
... # doctest: +NORMALIZE_WHITESPACE
Traceback (most recent call last):
...
TypeError: The adapter factory doesn't implement a single interface and
 no provided interface was specified.
>>> components.unregisterSubscriptionAdapter(tests.A, required=[tests.IA1])
... # doctest: +NORMALIZE_WHITESPACE
Traceback (most recent call last):
...
TypeError: The adapter factory doesn't implement a single interface
and no provided interface was specified.
```

If you unregister something that's not registered, nothing will be changed and False will be returned:

```
>>> components.unregisterSubscriptionAdapter(
...      required=[tests.I1], provided=tests.IA2)
False
```

Handlers

Handlers are used when you want to perform some function in response to an event. Handlers aren't expected to return anything when called and are not registered to provide any interface.

```
>>> from zope import component
>>> @component.adapter(tests.I1)
... def handle1(x):
...      print 'handle1', x
>>> components.registerHandler(handle1, info="First handler")
... # doctest: +NORMALIZE_WHITESPACE
Registered event:
HandlerRegistration(<Components comps>, [I1], u'',
                    handle1, 'First handler')
>>> components.handle(tests.U1(1))
handle1 U1(1)
>>> @component.adapter(tests.I1, tests.I2)
... def handle12(x, y):
...      print 'handle12', x, y
>>> components.registerHandler(handle12)
Registered event:
HandlerRegistration(<Components comps>, [I1, I2], u'', handle12, u'')
>>> components.handle(tests.U1(1), tests.U12(2))
handle12 U1(1) U12(2)
```

If a handler doesn't document interfaces it handles, then the required interfaces must be specified:

```
>>> def handle(*objects):
...     print 'handle', objects
>>> components.registerHandler(handle)
... # doctest: +NORMALIZE_WHITESPACE
Traceback (most recent call last):
...
TypeError: The adapter factory doesn't have a __component_adapts__
attribute and no required specifications were specified
>>> components.registerHandler(handle, required=[tests.I1],
...                            info="a comment")
Registered event:
HandlerRegistration(<Components comps>, [I1], u'', handle, 'a comment')
```

Handlers can also be registered for classes:

```
>>> components.registerHandler(handle, required=[tests.U],
...                            info="handle a class")
... # doctest: +NORMALIZE_WHITESPACE
Registered event:
HandlerRegistration(<Components comps>, [zope.component.tests.U], u'',
                    handle, 'handle a class')
>>> components.handle(tests.U1(1))
handle (U1(1),)
handle1 U1(1)
handle (U1(1),)
```

We can list the handler registrations:

```
>>> for registration in components.registeredHandlers():
...     print registration.required
...     print registration.handler, registration.info
... # doctest: +NORMALIZE_WHITESPACE
(<InterfaceClass zope.component.tests.I1>,)
<function handle1 at 0xb78f5bfc> First handler
(<InterfaceClass zope.component.tests.I1>,
 <InterfaceClass zope.component.tests.I2>)
<function handle12 at 0xb78f5c34>
(<InterfaceClass zope.component.tests.I1>,)
<function handle at 0xb78f5ca4> a comment
(<implementedBy zope.component.tests.U>,)
<function handle at 0xb78f5ca4> handle a class
```

and we can unregister handlers:

```
>>> components.unregisterHandler(required=[tests.U])
... # doctest: +NORMALIZE_WHITESPACE
Unregistered event:
HandlerRegistration(<Components comps>, [zope.component.tests.U], u'',
                    None, '')
True
>>> for registration in components.registeredHandlers():
...     print registration.required
...     print registration.handler, registration.info
... # doctest: +NORMALIZE_WHITESPACE
(<InterfaceClass zope.component.tests.I1>,)
<function handle1 at 0xb78f5bfc> First handler
(<InterfaceClass zope.component.tests.I1>,
 <InterfaceClass zope.component.tests.I2>)
```

```
<function handle12 at 0xb78f5c34>
(<InterfaceClass zope.component.tests.I1>,)
<function handle at 0xb78f5ca4> a comment
>>> components.unregisterHandler(handle12)
Unregistered event:
HandlerRegistration(<Components comps>, [I1, I2], u'', handle12, '')
True
>>> for registration in components.registeredHandlers():
...     print registration.required
...     print registration.handler, registration.info
(<InterfaceClass zope.component.tests.I1>,)
<function handle1 at 0xb78f5bfc> First handler
(<InterfaceClass zope.component.tests.I1>,)
<function handle at 0xb78f5ca4> a comment
>>> components.unregisterHandler(handle12)
False
>>> components.unregisterHandler()
Traceback (most recent call last):
...
TypeError: Must specify one of factory and required
>>> components.registerHandler(handle)
... # doctest: +NORMALIZE_WHITESPACE
Traceback (most recent call last):
...
TypeError: The adapter factory doesn't have a __component_adapts__
attribute and no required specifications were specified
```

Extending

Component-management objects can extend other component-management objects.

```
>>> c1 = registry.Components('1')
>>> c1.__bases__
()
>>> c2 = registry.Components('2', (c1, ))
>>> c2.__bases__ == (c1, )
True
>>> c1.registerUtility(tests.U1(1))
Registered event:
UtilityRegistration(<Components 1>, I1, u'', 1, u'')
>>> c1.queryUtility(tests.I1)
U1(1)
>>> c2.queryUtility(tests.I1)
U1(1)
>>> c1.registerUtility(tests.U1(2))
Registered event:
UtilityRegistration(<Components 1>, I1, u'', 2, u'')
>>> c2.queryUtility(tests.I1)
U1(2)
```

We can use multiple inheritence:

```
>>> c3 = registry.Components('3', (c1, ))
>>> c4 = registry.Components('4', (c2, c3))
>>> c4.queryUtility(tests.I1)
U1(2)
```

```
>>> c1.registerUtility(tests.U12(1), tests.I2)
Registered event:
UtilityRegistration(<Components 1>, I2, u'', 1, u'')
>>> c4.queryUtility(tests.I2)
U12(1)
>>> c3.registerUtility(tests.U12(3), tests.I2)
Registered event:
UtilityRegistration(<Components 3>, I2, u'', 3, u'')
>>> c4.queryUtility(tests.I2)
U12(3)
>>> c1.registerHandler(handle1, info="First handler")
Registered event:
HandlerRegistration(<Components 1>, [I1], u'', handle1, 'First handler')
>>> c2.registerHandler(handle, required=[tests.U])
... # doctest: +NORMALIZE_WHITESPACE
Registered event:
HandlerRegistration(<Components 2>, [zope.component.tests.U], u'',
                    handle, u'')
>>> @component.adapter(tests.I1)
... def handle3(x):
...     print 'handle3', x
>>> c3.registerHandler(handle3)
Registered event:
HandlerRegistration(<Components 3>, [I1], u'', handle3, u'')
>>> @component.adapter(tests.I1)
... def handle4(x):
...     print 'handle4', x
>>> c4.registerHandler(handle4)
Registered event:
HandlerRegistration(<Components 4>, [I1], u'', handle4, u'')
>>> c4.handle(tests.U1(1))
handle1 U1(1)
handle3 U1(1)
handle (U1(1),)
handle4 U1(1)
```

Redispatch of registration events

Some handlers are available that, if registered, redispatch registration events to the objects being registered. They depend on being dispatched to by the object-event dispatcher:

```
>>> from zope import component
>>> import zope.component.event
>>> zope.component.getGlobalSiteManager().registerHandler(
...      zope.component.event.objectEventNotify)
... # doctest: +NORMALIZE_WHITESPACE
Registered event:
HandlerRegistration(<BaseGlobalComponents base>,
                    [IObjectEvent], u'', objectEventNotify, u'')
```

To see this, we'll first register a multi-handler to show is when handlers are called on 2 objects:

```
>>> @zope.component.adapter(None, None)
... def double_handler(o1, o2):
...     print 'Double dispatch:'
...     print ' ', o1
...     print ' ', o2
>>> zope.component.getGlobalSiteManager().registerHandler(double_handler)
```

```
... # doctest: +NORMALIZE_WHITESPACE
Double dispatch:
  HandlerRegistration(<BaseGlobalComponents base>,
                      [Interface, Interface], u'', double_handler, u'')
  Registered event:
  HandlerRegistration(<BaseGlobalComponents base>,
  [Interface, Interface], u'', double_handler, u'')
Registered event:
HandlerRegistration(<BaseGlobalComponents base>,
                    [Interface, Interface], u'', double_handler, u'')
```

In the example above, the double_handler reported it's own registration. :)

Now we'll register our handlers:

```
>>> zope.component.getGlobalSiteManager().registerHandler(
...     registry.dispatchUtilityRegistrationEvent)
... # doctest: +NORMALIZE_WHITESPACE +ELLIPSIS
Double dispatch:
...
>>> zope.component.getGlobalSiteManager().registerHandler(
...     registry.dispatchAdapterRegistrationEvent)
... # doctest: +NORMALIZE_WHITESPACE +ELLIPSIS
Double dispatch:
...
>>> zope.component.getGlobalSiteManager().registerHandler(
...     registry.dispatchSubscriptionAdapterRegistrationEvent)
... # doctest: +NORMALIZE_WHITESPACE +ELLIPSIS
Double dispatch:
...
>>> zope.component.getGlobalSiteManager().registerHandler(
...     registry.dispatchHandlerRegistrationEvent)
... # doctest: +NORMALIZE_WHITESPACE
Double dispatch:
  HandlerRegistration(<BaseGlobalComponents base>,
                      [IHandlerRegistration, IRegistrationEvent], u'',
                      dispatchHandlerRegistrationEvent, u'')
  Registered event:
  HandlerRegistration(<BaseGlobalComponents base>,
                      [IHandlerRegistration, IRegistrationEvent], u'',
                      dispatchHandlerRegistrationEvent, u'')
Double dispatch:
  <function dispatchHandlerRegistrationEvent at 0xb799f72c>
  Registered event:
  HandlerRegistration(<BaseGlobalComponents base>,
                      [IHandlerRegistration, IRegistrationEvent], u'',
                      dispatchHandlerRegistrationEvent, u'')
Registered event:
HandlerRegistration(<BaseGlobalComponents base>,
                    [IHandlerRegistration, IRegistrationEvent], u'',
                    dispatchHandlerRegistrationEvent, u'')
```

In the last example above, we can see that the registration of dispatchHandlerRegistrationEvent was handled by dispatchHandlerRegistrationEvent and redispatched. This can be seen in the second double-dispatch output, where the first argument is the object being registered, which is dispatchHandlerRegistrationEvent.

If we change some other registrations, we can the double dispatch taking place:

```
>>> components.registerUtility(u5)
... # doctest: +NORMALIZE_WHITESPACE
```

```
Double dispatch:
  UtilityRegistration(<Components comps>, I1, u'', 5, u'')
  Registered event:
  UtilityRegistration(<Components comps>, I1, u'', 5, u'')
Double dispatch:
  U1(5)
  Registered event:
  UtilityRegistration(<Components comps>, I1, u'', 5, u'')
Registered event:
UtilityRegistration(<Components comps>, I1, u'', 5, u'')
>>> components.registerAdapter(tests.A12_1)
... # doctest: +NORMALIZE_WHITESPACE
Double dispatch:
  AdapterRegistration(<Components comps>, [I1, I2], IA1, u'', A12_1, u'')
  Registered event:
  AdapterRegistration(<Components comps>, [I1, I2], IA1, u'', A12_1, u'')
Double dispatch:
  zope.component.tests.A12_1
  Registered event:
  AdapterRegistration(<Components comps>, [I1, I2], IA1, u'', A12_1, u'')
Registered event:
AdapterRegistration(<Components comps>, [I1, I2], IA1, u'', A12_1, u'')
>>> components.registerSubscriptionAdapter(tests.A1_2)
... # doctest: +NORMALIZE_WHITESPACE
Double dispatch:
  SubscriptionRegistration(<Components comps>, [I1], IA2, u'', A1_2, u'')
  Registered event:
  SubscriptionRegistration(<Components comps>, [I1], IA2, u'', A1_2, u'')
Double dispatch:
  zope.component.tests.A1_2
  Registered event:
  SubscriptionRegistration(<Components comps>, [I1], IA2, u'', A1_2, u'')
Registered event:
SubscriptionRegistration(<Components comps>, [I1], IA2, u'', A1_2, u'')
```

Demo Demo