plone.app.testing Documentation

Release 4.2.1

Plone Foundation

August 04, 2015

Contents

CHAPTER 1

Introduction

Table of contents

`plone.app.testing` provides tools for writing integration and functional tests for code that runs on top of Plone. It is based on plone.testing. If you are unfamiliar with `plone.testing`, the concept of layers, or the zope.testing testrunner, please take a look at the the `plone.testing` documentation. In fact, even if you are working exclusively with Plone, you are likely to want to use some of its features for unit testing.

In short, `plone.app.testing` includes:

- A set of layers that set up fixtures containing a Plone site, intended for writing integration and functional tests.
- A collection of helper functions, some useful for writing your own layers and some applicable to tests themselves.
- A convenient layer base class, extending `plone.testing.Layer`, which makes it easier to write custom layers extending the Plone site fixture, with proper isolation and tear-down.
- Cleanup hooks for `zope.testing.cleanup` to clean up global state found in a Plone installation. This is useful for unit testing.

1.1 Compatibility

`plone.app.testing` 4.x works with Plone 4 and Zope 2.12. It may work with newer versions. It will not work with earlier versions. Use `plone.app.testing` 3.x for Plone 3 and Zope 2.10.

CHAPTER 2

Installation and usage

To use `plone.app.testing` in your own package, you need to add it as a dependency. Most people prefer to keep test-only dependencies separate, so that they do not need to be installed in scenarios (such as on a production server) where the tests will not be run. This can be achieved using a `test` extra.

In `setup.py`, add or modify the `extras_require` option, like so:

```
extras_require = {
    'test': [
            'plone.app.testing',
        ]
},
```

This will also include `plone.testing`, with the `[z2]`, `[zca]` and `[zodb]` extras (which `plone.app.testing` itself relies on).

Please see the plone.testing documentation for more details about how to add a test runner to your buildout, and how to write and run tests.

CHAPTER 3

Layer reference

This package contains a layer class, `plone.app.testing.layers.PloneFixture`, which sets up a Plone site fixture. It is combined with other layers from plone.testing to provide a number of layer instances. It is important to realise that these layers all have the same fundamental fixture: they just manage test setup and tear-down differently.

When set up, the fixture will:

- Create a ZODB sandbox, via a stacked `DemoStorage`. This ensures persistent changes made during layer setup can be cleanly torn down.
- Configure a global component registry sandbox. This ensures that global component registrations (e.g. as a result of loading ZCML configuration) can be cleanly torn down.
- Create a configuration context with the `disable-autoinclude` feature set. This has the effect of stopping Plone from automatically loading the configuration of any installed package that uses the `z3c.autoinclude.plugin:plone` entry point via z3c.autoinclude. (This is to avoid accidentally polluting the test fixture - custom layers should load packages' ZCML configuration explicitly if required).
- Install a number of Zope 2-style products on which Plone depends.
- Load the ZCML for these products, and for `Products.CMFPlone`, which in turn pulls in the configuration for the core of Plone.
- Create a default Plone site, with the default theme enabled, but with no default content.
- Add a user to the root user folder with the `Manager` role.
- Add a test user to this instance with the `Member` role.

For each test:

- The test user is logged in
- The local component site is set
- Various global caches are cleaned up

Various constants in the module `plone.app.testing.interfaces` are defined to describe this environment:

Constant	Purpose
PLONE_SITE_ID	The id of the Plone site object inside the Zope application root.
PLONE_SITE_TITLE	The title of the Plone site
DEFAULT_LANGUAGE	The default language of the Plone site ('en')
TEST_USER_ID	The id of the test user
TEST_USER_NAME	The username of the test user
TEST_USER_PASSWORD	The password of the test user
TEST_USER_ROLES	The default global roles of the test user - ('Member',)
SITE_OWNER_NAME	The username of the user owning the Plone site.
SITE_OWNER_PASSWORD	The password of the user owning the Plone site.

All the layers also expose a resource in addition to those from their base layers, made available during tests:

`portal` The Plone site root.

3.1 Plone site fixture

Layer:	`plone.app.testing.PLONE_FIXTURE`
Class:	`plone.app.testing.layers.PloneFixture`
Bases:	`plone.testing.z2.STARTUP`
Resources:	

This layer sets up the Plone site fixture on top of the `z2.STARTUP` fixture.

You should not use this layer directly, as it does not provide any test lifecycle or transaction management. Instead, you should use a layer created with either the `IntegrationTesting` or `FunctionalTesting` classes, as outlined below.

3.2 Integration and functional testing test lifecycles

`plone.app.testing` comes with two layer classes, `IntegrationTesting` and `FunctionalTesting`, which derive from the corresponding layer classes in `plone.testing.z2`.

These classes set up the `app`, `request` and `portal` resources, and reset the fixture (including various global caches) between each test run.

As with the classes in `plone.testing`, the `IntegrationTesting` class will create a new transaction for each test and roll it back on test tear- down, which is efficient for integration testing, whilst `FunctionalTesting` will create a stacked `DemoStorage` for each test and pop it on test tear- down, making it possible to exercise code that performs an explicit commit (e.g. via tests that use `zope.testbrowser`).

When creating a custom fixture, the usual pattern is to create a new layer class that has `PLONE_FIXTURE` as its default base, instantiating that as a separate "fixture" layer. This layer is not to be used in tests directly, since it won't have test/transaction lifecycle management, but represents a shared fixture, potentially for both functional and integration testing. It is also the point of extension for other layers that follow the same pattern.

Once this fixture has been defined, "end-user" layers can be defined using the `IntegrationTesting` and `FunctionalTesting` classes. For example:

```
from plone.testing import Layer
from plone.app.testing import PLONE_FIXTURE
from plone.app.testing import IntegrationTesting, FunctionalTesting

class MyFixture(Layer):
    defaultBases = (PLONE_FIXTURE,)
```

```
    ...

MY_FIXTURE = MyFixture()

MY_INTEGRATION_TESTING = IntegrationTesting(bases=(MY_FIXTURE,), name="MyFixture:Integration")
MY_FUNCTIONAL_TESTING = FunctionalTesting(bases=(MY_FIXTURE,), name="MyFixture:Functional")
```

See the `PloneSandboxLayer` layer below for a more comprehensive example.

3.3 Plone integration testing

Layer:	`plone.app.testing.PLONE_INTEGRATION_TESTING`
Class:	`plone.app.testing.layers.IntegrationTesting`
Bases:	`plone.app.testing.PLONE_FIXTURE`
Resources:	`portal` (test setup only)

This layer can be used for integration testing against the basic `PLONE_FIXTURE` layer.

You can use this directly in your tests if you do not need to set up any other shared fixture.

However, you would normally not extend this layer - see above.

3.4 Plone functional testing

Layer:	`plone.app.testing.PLONE_FUNCTIONAL_TESTING`
Class:	`plone.app.testing.layers.FunctionalTesting`
Bases:	`plone.app.testing.PLONE_FIXTURE`
Resources:	`portal` (test setup only)

This layer can be used for functional testing against the basic `PLONE_FIXTURE` layer, for example using `zope.testbrowser`.

You can use this directly in your tests if you do not need to set up any other shared fixture.

Again, you would normally not extend this layer - see above.

3.5 Plone ZServer

Layer:	`plone.app.testing.PLONE_ZSERVER`
Class:	`plone.testing.z2.ZServer`
Bases:	`plone.app.testing.PLONE_FUNCTIONAL_TESTING`
Resources:	`portal` (test setup only)

This is layer is intended for functional testing using a live, running HTTP server, e.g. using Selenium or Windmill.

Again, you would not normally extend this layer. To create a custom layer that has a running ZServer, you can use the same pattern as this one, e.g.:

```
from plone.testing import Layer
from plone.testing import z2
from plone.app.testing import PLONE_FIXTURE
from plone.app.testing import FunctionalTesting
```

```
class MyFixture(Layer):
    defaultBases = (PLONE_FIXTURE,)

    ...

MY_FIXTURE = MyFixture()
MY_ZSERVER = FunctionalTesting(bases=(MY_FIXTURE, z2.ZSERVER_FIXTURE), name='MyFixture:ZServer')
```

See the description of the `z2.ZSERVER` layer in plone.testing for further details.

3.6 Plone FTP server

Layer:	`plone.app.testing.PLONE_FTP_SERVER`
Class:	`plone.app.testing.layers.FunctionalTesting`
Bases:	`plone.app.testing.PLONE_FIXTURE` `plone.testing.z2.ZSERVER_FIXTURE`
Resources:	`portal` (test setup only)

This is layer is intended for functional testing using a live FTP server.

It is semantically equivalent to the `PLONE_ZSERVER` layer.

See the description of the `z2.FTP_SERVER` layer in plone.testing for further details.

CHAPTER 4

Helper functions

A number of helper functions are provided for use in tests and custom layers.

4.1 Plone site context manager

ploneSite(db=None, connection=None, environ=None) Use this context manager to access and make changes to the Plone site during layer setup. In most cases, you will use it without arguments, but if you have special needs, you can tie it to a particular database instance. See the description of the `zopeApp()` context manager in plone.testing (which this context manager uses internally) for details.

The usual pattern is to call it during `setUp()` or `tearDown()` in your own layers:

```
from plone.testing import Layer
from plone.app.testing import ploneSite

class MyLayer(Layer):

    def setUp(self):

        ...

        with ploneSite() as portal:

            # perform operations on the portal, e.g.
            portal.title = u"New title"
```

Here, `portal` is the Plone site root. A transaction is begun before entering the `with` block, and will be committed upon exiting the block, unless an exception is raised, in which case it will be rolled back.

Inside the block, the local component site is set to the Plone site root, so that local component lookups should work.

Warning: Do not attempt to load ZCML files inside a `ploneSite` block. Because the local site is set to the Plone site, you may end up accidentally registering components in the local site manager, which can cause pickling errors later.

Note: You should not use this in a test, or in a `testSetUp()` or `testTearDown()` method of a layer based on one of the layer in this package. Use the `portal` resource instead.

Also note: If you are writing a layer setting up a Plone site fixture, you may want to use the `PloneSandboxLayer` layer base class, and implement the `setUpZope()`, `setUpPloneSite()`, `tearDownZope()` and/or `tearDownPloneSite()` methods instead. See below.

4.2 User management

login(portal, userName) Simulate login as the given user. This is based on the `z2.login()` helper in plone.testing, but instead of passing a specific user folder, you pass the portal (e.g. as obtained via the `portal` layer resource).

For example:

```
import unittest2 as unittest

from plone.app.testing import PLONE_INTEGRATION_TESTING
from plone.app.testing import TEST_USER_NAME
from plone.app.testing import login

...

class MyTest(unittest.TestCase):

    layer = PLONE_INTEGRATION_TESTING

    def test_something(self):
        portal = self.layer['portal']
        login(portal, TEST_USER_NAME)

        ...
```

logout() Simulate logging out, i.e. becoming the anonymous user. This is equivalent to the `z2.logout()` helper in plone.testing.

For example:

```
import unittest2 as unittest

from plone.app.testing import PLONE_INTEGRATION_TESTING
from plone.app.testing import logout

...

class MyTest(unittest.TestCase):

    layer = PLONE_INTEGRATION_TESTING

    def test_something(self):
        portal = self.layer['portal']
        logout()

        ...
```

setRoles(portal, userId, roles) Set the roles for the given user. `roles` is a list of roles.

For example:

```
import unittest2 as unittest

from plone.app.testing import PLONE_INTEGRATION_TESTING
from plone.app.testing import TEST_USER_ID
from plone.app.testing import setRoles

...
```

```
    class MyTest(unittest.TestCase):

        layer = PLONE_INTEGRATION_TESTING

        def test_something(self):
            portal = self.layer['portal']
            setRoles(portal, TEST_USER_ID, ['Manager'])
```

4.3 Product and profile installation

applyProfile(portal, profileName) Install a GenericSetup profile (usually an extension profile) by name, using the `portal_setup` tool. The name is normally made up of a package name and a profile name. Do not use the `profile-` prefix.

For example:

```
    from plone.testing import Layer

    from plone.app.testing import ploneSite
    from plone.app.testing import applyProfile

    ...

    class MyLayer(Layer):

        ...

        def setUp(self):

            ...

            with ploneSite() as portal:
                applyProfile(portal, 'my.product:default')

                ...
```

quickInstallProduct(portal, productName, reinstall=False) Use this function to install a particular product into the given Plone site, using the `portal_quickinstaller` tool. If `reinstall` is `False` and the product is already installed, nothing will happen; if `reinstall` is `True`, the product will be reinstalled. The `productName` should be a full dotted name, e.g. `Products.MyProduct`, or `my.product`.

For example:

```
    from plone.testing import Layer

    from plone.app.testing import ploneSite
    from plone.app.testing import quickInstallProduct

    ...

    class MyLayer(Layer):

        ...

        def setUp(self):
```

```
        ...

        with ploneSite() as portal:
            quickInstallProduct(portal, 'my.product')

            ...
```

4.4 Component architecture sandboxing

pushGlobalRegistry(portal, new=None, name=None) Create or obtain a stack of global component registries, and push a new registry to the top of the stack. This allows Zope Component Architecture registrations (e.g. loaded via ZCML) to be effectively torn down.

If you are going to use this function, please read the corresponding documentation for `zca.pushGlobalRegistry()` in plone.testing. In particular, note that you *must* reciprocally call `popGlobalRegistry()` (see below).

This helper is based on `zca.pushGlobalRegistry()`, but will also fix up the local component registry in the Plone site `portal` so that it has the correct bases.

For example:

```
    from plone.testing import Layer

    from plone.app.testing import ploneSite
    from plone.app.testing import pushGlobalRegistry
    from plone.app.testing import popGlobalRegistry

    ...

    class MyLayer(Layer):

        ...

        def setUp(self):

            ...

            with ploneSite() as portal:
                pushGlobalRegistry(portal)

                ...
```

popGlobalRegistry(portal) Tear down the top of the component architecture stack, as created with `pushGlobalRegistry()`

For example:

```
    ...

        def tearDown(self):

            with ploneSite() as portal:
                popGlobalRegistry(portal)
```

4.5 Global state cleanup

tearDownMultiPluginRegistration(pluginName) PluggableAuthService "MultiPlugins" are kept in a global registry. If you have registered a plugin, e.g. using the `registerMultiPlugin()` API, you should tear that registration down in your layer's `tearDown()` method. You can use this helper, passing a plugin name.

For example:

```
from plone.testing import Layer

from plone.app.testing import ploneSite
from plone.app.testing import tearDownMultiPluginRegistration

...

class MyLayer(Layer):

    ...

    def tearDown(self):

        tearDownMultiPluginRegistration('MyPlugin')

        ...
```

CHAPTER 5

Layer base class

If you are writing a custom layer to test your own Plone add-on product, you will often want to do the following on setup:

1. Stack a new `DemoStorage` on top of the one from the base layer. This ensures that any persistent changes performed during layer setup can be torn down completely, simply by popping the demo storage.
2. Stack a new ZCML configuration context. This keeps separate the information about which ZCML files were loaded, in case other, independent layers want to load those same files after this layer has been torn down.
3. Push a new global component registry. This allows you to register components (e.g. by loading ZCML or using the test API from `zope.component`) and tear down those registration easily by popping the component registry.
4. Load your product's ZCML configuration
5. Install the product into the test fixture Plone site

Of course, you may wish to make other changes too, such as creating some base content or changing some settings.

On tear-down, you will then want to:

1. Remove any Pluggable Authentication Service "multi-plugins" that were added to the global registry during setup.
2. Pop the global component registry to unregister components loaded via ZCML.
3. Pop the configuration context resource to restore its state.
4. Pop the `DemoStorage` to undo any persistent changes.

If you have made other changes on setup that are not covered by this broad tear-down, you'll also want to tear those down explicitly here.

Stacking a demo storage and component registry is the safest way to avoid fixtures bleeding between tests. However, it can be tricky to ensure that everything happens in the right order.

To make things easier, you can use the `PloneSandboxLayer` layer base class. This extends `plone.testing.Layer` and implements `setUp()` and `tearDown()` for you. You simply have to override one or more of the following methods:

`setUpZope(self, app, configurationContext)` This is called during setup. `app` is the Zope application root. `configurationContext` is a newly stacked ZCML configuration context. Use this to load ZCML, install products using the helper `plone.testing.z2.installProduct()`, or manipulate other global state.

setUpPloneSite(self, portal) This is called during setup. `portal` is the Plone site root as configured by the `ploneSite()` context manager. Use this to make persistent changes inside the Plone site, such as installing products using the `applyProfile()` or `quickInstallProduct()` helpers, or setting up default content.

tearDownZope(self, app) This is called during tear-down, before the global component registry and stacked `DemoStorage` are popped. Use this to tear down any additional global state.

Note: Global component registrations PAS multi-plugin registrations are automatically torn down. Product installations are not, so you should use the `uninstallProduct()` helper if any products were installed during `setUpZope()`.

tearDownPloneSite(self, portal) This is called during tear-down, before the global component registry and stacked `DemoStorage` are popped. During this method, the local component site hook is set, giving you access to local components.

Note: Persistent changes to the ZODB are automatically torn down by virtue of a stacked `DemoStorage`. Thus, this method is less commonly used than the others described here.

Let's show a more comprehensive example of what such a layer may look like. Imagine we have a product `my.product`. It has a `configure.zcml` file that loads some components and registers a `GenericSetup` profile, making it installable in the Plone site. On layer setup, we want to load the product's configuration and install it into the Plone site.

The layer would conventionally live in a module `testing.py` at the root of the package, i.e. `my.product.testing`:

```
from plone.app.testing import PloneSandboxLayer
from plone.app.testing import PLONE_FIXTURE
from plone.app.testing import IntegrationTesting

from plone.testing import z2

class MyProduct(PloneSandboxLayer):

    defaultBases = (PLONE_FIXTURE,)

    def setUpZope(self, app, configurationContext):
        # Load ZCML
        import my.product
        self.loadZCML(package=my.product)

        # Install product and call its initialize() function
        z2.installProduct(app, 'my.product')

        # Note: you can skip this if my.product is not a Zope 2-style
        # product, i.e. it is not in the Products.* namespace and it
        # does not have a <five:registerPackage /> directive in its
        # configure.zcml.

    def setUpPloneSite(self, portal):
        # Install into Plone site using portal_setup
        self.applyProfile(portal, 'my.product:default')

    def tearDownZope(self, app):
        # Uninstall product
        z2.uninstallProduct(app, 'my.product')

        # Note: Again, you can skip this if my.product is not a Zope 2-
        # style product
```

```
MY_PRODUCT_FIXTURE = MyProduct()
MY_PRODUCT_INTEGRATION_TESTING = IntegrationTesting(bases=(MY_PRODUCT_FIXTURE,), name="MyProduct:Inte
```

Here, `MY_PRODUCT_FIXTURE` is the "fixture" base layer. Other layers can use this as a base if they want to build on this fixture, but it would not be used in tests directly. For that, we have crated an `IntegrationTesting` instance, `MY_PRODUCT_INTEGRATION_TESTING`.

Of course, we could have created a `FunctionalTesting` instance as well, e.g.:

```
MY_PRODUCT_FUNCTIONAL_TESTING = FunctionalTesting(bases=(MY_PRODUCT_FIXTURE,), name="MyProduct:Functi
```

Of course, we could do a lot more in the layer setup. For example, let's say the product had a content type 'my.product.page' and we wanted to create some test content. We could do that with:

```
from plone.app.testing import TEST_USER_ID
from plone.app.testing import TEST_USER_NAME
from plone.app.testing import login
from plone.app.testing import setRoles

...

    def setUpPloneSite(self, portal):

        ...

        setRoles(portal, TEST_USER_ID, ['Manager'])
        login(portal, TEST_USER_NAME)
        portal.invokeFactory('my.product.page', 'page-1', title=u"Page 1")
        setRoles(portal, TEST_USER_ID, ['Member'])

...
```

Note that unlike in a test, there is no user logged in at layer setup time, so we have to explicitly log in as the test user. Here, we also grant the test user the `Manager` role temporarily, to allow object construction (which performs an explicit permission check).

> **Note:** Automatic tear down suffices for all the test setup above. If the only changes made during layer setup are to persistent, in-ZODB data, or the global component registry then no additional tear-down is required. For any other global state being managed, you should write a `tearDownPloneSite()` method to perform the necessary cleanup.

Given this layer, we could write a test (e.g. in `tests.py`) like:

```
import unittest2 as unittest
from my.product.testing import MY_PRODUCT_INTEGRATION_TESTING

class IntegrationTest(unittest.TestCase):

    layer = MY_PRODUCT_INTEGRATION_TESTING

    def test_page_dublin_core_title(self):
        portal = self.layer['portal']

        page1 = portal['page-1']
        page1.title = u"Some title"

        self.assertEqual(page1.Title(), u"Some title")
```

Please see plone.testing for more information about how to write and run tests and assertions.

CHAPTER 6

Common test patterns

plone.testing's documentation contains details about the fundamental techniques for writing tests of various kinds. In a Plone context, however, some patterns tend to crop up time and again. Below, we will attempt to catalogue some of the more commonly used patterns via short code samples.

The examples in this section are all intended to be used in tests. Some may also be useful in layer set-up/tear-down. We have used `unittest` syntax here, although most of these examples could equally be adopted to doctests.

We will assume that you are using a layer that has `PLONE_FIXTURE` as a base (whether directly or indirectly) and uses the `IntegrationTesting` or `FunctionalTesting` classes as shown above.

We will also assume that the variables `app`, `portal` and `request` are defined from the relative layer resources, e.g. with:

```
app = self.layer['app']
portal = self.layer['portal']
request = self.layer['request']
```

Note that in a doctest set up using the `layered()` function from `plone.testing`, `layer` is in the global namespace, so you would do e.g. `portal = layer['portal']`.

Where imports are required, they are shown alongside the code example. If a given import or variable is used more than once in the same section, it will only be shown once.

6.1 Basic content management

To create a content item of type 'Folder' with the id 'f1' in the root of the portal:

```
portal.invokeFactory('Folder', 'f1', title=u"Folder 1")
```

The `title` argument is optional. Other basic properties, like `description`, can be set as well.

Note that this may fail with an `Unauthorized` exception, since the test user won't normally have permissions to add content in the portal root, and the `invokeFactory()` method performs an explicit security check. You can set the roles of the test user to ensure that he has the necessary permissions:

```
from plone.app.testing import setRoles
from plone.app.testing import TEST_USER_ID

setRoles(portal, TEST_USER_ID, ['Manager'])
portal.invokeFactory('Folder', 'f1', title=u"Folder 1")
```

To obtain this object, acquisition-wrapped in its parent:

```
f1 = portal['f1']
```

To make an assertion against an attribute or method of this object:

```
self.assertEqual(f1.Title(), u"Folder 1")
```

To modify the object:

```
f1.setTitle(u"Some title")
```

To add another item inside the folder f1:

```
f1.invokeFactory('Document', 'd1', title=u"Document 1")
d1 = f1['d1']
```

To check if an object is in a container:

```
self.assertTrue('f1' in portal)
```

To delete an object from a container:

del portal['f1']

There is no content or workflows installed by default. You can enable workflows:

```
portal.portal_workflow.setDefaultChain("simple_publication_workflow")
```

6.2 Searching

To obtain the `portal_catalog` tool:

```
from Products.CMFCore.utils import getToolByName

catalog = getToolByName(portal, 'portal_catalog')
```

To search the catalog:

```
results = catalog(portal_type="Document")
```

Keyword arguments are search parameters. The result is a lazy list. You can call `len()` on it to get the number of search results, or iterate through it. The items in the list are catalog brains. They have attributes that correspond to the "metadata" columns configured for the catalog, e.g. `Title`, `Description`, etc. Note that these are simple attributes (not methods), and contain the value of the corresponding attribute or method from the source object at the time the object was cataloged (i.e. they are not necessarily up to date).

To make assertions against the search results:

```
self.assertEqual(len(results), 1)

# Copy the list into memory so that we can use [] notation
results = list(results)

# Check the first (and in this case only) result in the list
self.assertEqual(results[0].Title, u"Document 1")
```

To get the path of a given item in the search results:

```
self.assertEqual(resuls[0].getPath(), portal.absolute_url_path() + '/f1/d1')
```

To get an absolute URL:

```
self.assertEqual(resuls[0].getURL(), portal.absolute_url() + '/f1/d1')
```

To get the original object:

```
obj = results[0].getObject()
```

To re-index an object d1 so that its catalog information is up to date:

```
d1.reindexObject()
```

6.3 User management

To create a new user:

```
from Products.CMFCore.utils import getToolByName

acl_users = getToolByName(portal, 'acl_users')

acl_users.userFolderAddUser('user1', 'secret', ['Member'], [])
```

The arguments are the username (which will also be the user id), the password, a list of roles, and a list of domains (rarely used).

To make a particular user active ("logged in") in the integration testing environment use the `login` method and pass it the username:

```
from plone.app.testing import login

login(portal, 'user1')
```

To log out (become anonymous):

```
from plone.app.testing import logout

logout()
```

To obtain the current user:

```
from AccessControl import getSecurityManager

user = getSecurityManager().getUser()
```

To obtain a user by name:

```
user = acl_users.getUser('user1')
```

Or by user id (id and username are often the same, but can differ in real-world scenarios):

```
user = acl_users.getUserById('user1')
```

To get the user's user name:

```
userName = user.getUserName()
```

To get the user's id:

```
userId = user.getId()
```

6.4 Permissions and roles

To get a user's roles in a particular context (taking local roles into account):

```
from AccessControl import getSecurityManager

user = getSecurityManager().getUser()

self.assertEqual(user.getRolesInContext(portal), ['Member'])
```

To change the test user's roles:

```
from plone.app.testing import setRoles
from plone.app.testing import TEST_USER_ID

setRoles(portal, TEST_USER_ID, ['Member', 'Manager'])
```

Pass a different user name to change the roles of another user.

To grant local roles to a user in the folder f1:

```
f1.manage_setLocalRoles(TEST_USER_ID, ('Reviewer',))
```

To check the local roles of a given user in the folder 'f1':

```
self.assertEqual(f1.get_local_roles_for_userid(TEST_USER_ID), ('Reviewer',))
```

To grant the 'View' permission to the roles 'Member' and 'Manager' in the portal root without acquiring additional roles from its parents:

```
portal.manage_permission('View', ['Member', 'Manager'], acquire=False)
```

This method can also be invoked on a folder or individual content item.

To assert which roles have the permission 'View' in the context of the portal:

```
roles = [r['name'] for r in portal.rolesOfPermission('View') if r['selected']]
self.assertEqual(roles, ['Member', 'Manager'])
```

To assert which permissions have been granted to the 'Reviewer' role in the context of the portal:

```
permissions = [p['name'] for p in portal.permissionsOfRole('Reviewer') if p['selected']]
self.assertTrue('Review portal content' in permissions)
```

To add a new role:

```
portal._addRole('Tester')
```

This can now be assigned to users globally (using the `setRoles` helper) or locally (using `manage_setLocalRoles()`).

To assert which roles are available in a given context:

```
self.assertTrue('Tester' in portal.valid_roles())
```

6.5 Workflow

To set the default workflow chain:

```
from Products.CMFCore.utils import getToolByName

workflowTool = getToolByName(portal, 'portal_workflow')

workflowTool.setDefaultChain('my_workflow')
```

In Plone, most chains contain only one workflow, but the `portal_workflow` tool supports longer chains, where an item is subject to more than one workflow simultaneously.

To set a multi-workflow chain, separate workflow names by commas.

To get the default workflow chain:

```
self.assertEqual(workflowTool.getDefaultChain(), ('my_workflow',))
```

To set the workflow chain for the 'Document' type:

```
workflowTool.setChainForPortalTypes(('Document',), 'my_workflow')
```

You can pass multiple type names to set multiple chains at once. To set a multi-workflow chain, separate workflow names by commas. To indicate that a type should use the default workflow, use the special chain name '(Default)'.

To get the workflow chain for the portal type 'Document':

```
chains = dict(workflowTool.listChainOverrides())
defaultChain = workflowTool.getDefaultChain()
documentChain = chains.get('Document', defaultChain)

self.assertEqual(documentChain, ('my_other_workflow',))
```

To get the current workflow chain for the content object f1:

```
self.assertEqual(workflowTool.getChainFor(f1), ('my_workflow',))
```

To update all permissions after changing the workflow:

```
workflowTool.updateRoleMappings()
```

To change the workflow state of the content object f1 by invoking the transaction 'publish':

```
workflowTool.doActionFor(f1, 'publish')
```

Note that this performs an explicit permission check, so if the current user doesn't have permission to perform this workflow action, you may get an error indicating the action is not available. If so, use `login()` or `setRoles()` to ensure the current user is able to change the workflow state.

To check the current workflow state of the content object f1:

```
self.assertEqual(workflowTool.getInfoFor(f1, 'review_state'), 'published')
```

6.6 Properties

To set the value of a property on the portal root:

```
portal._setPropValue('title', u"My title")
```

To assert the value of a property on the portal root:

```
self.assertEqual(portal.getProperty('title'), u"My title")
```

To change the value of a property in a property sheet in the `portal_properties` tool:

```
from Products.CMFCore.utils import getToolByName

propertiesTool = getToolByName(portal, 'portal_properties')
siteProperties = propertiesTool['site_properties']

siteProperties._setPropValue('many_users', True)
```

To assert the value of a property in a property sheet in the `portal_properties` tool:

```
self.assertEqual(siteProperties.getProperty('many_users'), True)
```

6.7 Installing products and extension profiles

To apply a particular extension profile:

```
from plone.app.testing import applyProfile

applyProfile(portal, 'my.product:default')
```

This is the preferred method of installing a product's configuration.

To install an add-on product into the Plone site using the `portal_quickinstaller` tool:

```
from plone.app.testing import quickInstallProduct

quickInstallProduct(portal, 'my.product')
```

To re-install a product using the quick-installer:

```
quickInstallProduct(portal, 'my.product', reinstall=True)
```

Note that both of these assume the product's ZCML has been loaded, which is usually done during layer setup. See the layer examples above for more details on how to do that.

When writing a product that has an installation extension profile, it is often desirable to write tests that inspect the state of the site after the profile has been applied. Some of the more common such tests are shown below.

To verify that a product has been installed (e.g. as a dependency via `metadata.xml`):

```
from Products.CMFCore.utils import getToolByName

quickinstaller = getToolByName(portal, 'portal_quickinstaller')
self.assertTrue(quickinstaller.isProductInstalled('my.product'))
```

To verify that a particular content type has been installed (e.g. via `types.xml`):

```
typesTool = getToolByName(portal, 'portal_types')

self.assertNotEqual(typesTool.getTypeInfo('mytype'), None)
```

To verify that a new catalog index has been installed (e.g. via `catalog.xml`):

```
catalog = getToolByName(portal, 'portal_catalog')

self.assertTrue('myindex' in catalog.indexes())
```

To verify that a new catalog metadata column has been added (e.g. via `catalog.xml`):

```
self.assertTrue('myattr' in catalog.schema())
```

To verify that a new workflow has been installed (e.g. via `workflows.xml`):

```
workflowTool = getToolByName(portal, 'portal_workflow')

self.assertNotEqual(workflowTool.getWorkflowById('my_workflow'), None)
```

To verify that a new workflow has been assigned to a type (e.g. via `workflows.xml`):

```
self.assertEqual(dict(workflowTool.listChainOverrides())['mytype'], ('my_workflow',))
```

To verify that a new workflow has been set as the default (e.g. via `workflows.xml`):

```
self.assertEqual(workflowTool.getDefaultChain(), ('my_workflow',))
```

To test the value of a property in the `portal_properties` tool (e.g. set via `propertiestool.xml`)::

```
propertiesTool = getToolByName(portal, 'portal_properties')
siteProperties = propertiesTool['site_properties']

self.assertEqual(siteProperties.getProperty('some_property'), "some value")
```

To verify that a stylesheet has been installed in the `portal_css` tool (e.g. via `cssregistry.xml`):

```
cssRegistry = getToolByName(portal, 'portal_css')

self.assertTrue('mystyles.css' in cssRegistry.getResourceIds())
```

To verify that a JavaScript resource has been installed in the `portal_javascripts` tool (e.g. via `jsregistry.xml`):

```
jsRegistry = getToolByName(portal, 'portal_javascripts')

self.assertTrue('myscript.js' in jsRegistry.getResourceIds())
```

To verify that a new role has been added (e.g. via `rolemap.xml`):

```
self.assertTrue('NewRole' in portal.valid_roles())
```

To verify that a permission has been granted to a given set of roles (e.g. via `rolemap.xml`):

```
roles = [r['name'] for r in portal.rolesOfPermission('My Permission') if r['selected']]
self.assertEqual(roles, ['Member', 'Manager'])
```

6.8 Traversal

To traverse to a view, page template or other resource, use `restrictedTraverse()` with a relative path:

```
resource = portal.restrictedTraverse('f1/@@folder_contents')
```

The return value is a view object, page template object, or other resource. It may be invoked to obtain an actual response (see below).

`restrictedTraverse()` performs an explicit security check, and so may raise `Unauthorized` if the current test user does not have permission to view the given resource. If you don't want that, you can use:

```
resource = portal.unrestrictedTraverse('f1/@@folder_contents')
```

You can call this on a folder or other content item as well, to traverse from that starting point, e.g. this is equivalent to the first example above:

```
f1 = portal['f1']
resource = f1.restrictedTraverse('@@folder_contents')
```

Note that this traversal will not take `IPublishTraverse` adapters into account, and you cannot pass query string parameters. In fact, `restrictedTraverse()` and `unrestrictedTraverse()` implement the type of traversal that happens with path expressions in TAL, which is similar, but not identical to URL traversal.

To look up a view manually:

```
from zope.component import getMultiAdapter

view = getMultiAdapter((f1, request), name=u"folder_contents")
```

Note that the name here should not include the `@@` prefix.

To simulate an `IPublishTraverse` adapter call, presuming the view implements `IPublishTraverse`:

```
next = view.IPublishTraverse(request, u"some-name")
```

Or, if the `IPublishTraverse` adapter is separate from the view:

```
from zope.publisher.interfaces import IPublishTraverse

publishTraverse = getMultiAdapter((f1, request), IPublishTraverse)
next = view.IPublishTraverse(request, u"some-name")
```

To simulate a form submission or query string parameters:

```
request.form.update({
        'name': "John Smith",
        'age':  23
    })
```

The `form` dictionary contains the marshalled request. That is, if you are simulating a query string parameter or posted form variable that uses a marshaller like `:int` (e.g. `age:int` in the example above), the value in the `form` dictionary should be marshalled (an int instead of a string, in the example above), and the name should be the base name (`age` instead of `age:int`).

To invoke a view and obtain the response body as a string:

```
view = f1.restrictedTraverse('@@folder_contents')
body = view()

self.assertFalse(u"An unexpected error occurred" in body)
```

Please note that this approach is not perfect. In particular, the request is will not have the right URL or path information. If your view depends on this, you can fake it by setting the relevant keys in the request, e.g.:

```
request.set('URL', f1.absolute_url() + '/@@folder_contents')
request.set('ACTUAL_URL', f1.absolute_url() + '/@@folder_contents')
```

To inspect the state of the request (e.g. after a view has been invoked):

```
self.assertEqual(request.get('disable_border'), True)
```

To inspect response headers (e.g. after a view has been invoked):

```
response = request.response

self.assertEqual(response.getHeader('content-type'), 'text/plain')
```

6.9 Simulating browser interaction

End-to-end functional tests can use zope.testbrowser to simulate user interaction. This acts as a web browser, connecting to Zope via a special channel, making requests and obtaining responses.

> **Note:** zope.testbrowser runs entirely in Python, and does not simulate a JavaScript engine.

Note that to use `zope.testbrowser`, you need to use one of the functional testing layers, e.g. `PLONE_FUNCTIONAL_TESTING`, or another layer instantiated with the `FunctionalTesting` class.

If you want to create some initial content, you can do so either in a layer, or in the test itself, before invoking the test browser client. In the latter case, you need to commit the transaction before it becomes available, e.g.:

```
from plone.app.testing import setRoles
from plone.app.testing import TEST_USER_ID

# Make some changes
setRoles(portal, TEST_USER_ID, ['Manager'])
portal.invokeFactory('Folder', 'f1', title=u"Folder 1")
setRoles(portal, TEST_USER_ID, ['Member'])

# Commit so that the test browser sees these changes
import transaction
transaction.commit()
```

To obtain a new test browser client:

```
from plone.testing.z2 import Browser

browser = Browser(app)
```

To open a given URL:

```
portalURL = portal.absolute_url()
browser.open(portalURL)
```

To inspect the response:

```
self.assertTrue(u"Welcome" in browser.contents)
```

To inspect response headers:

```
self.assertEqual(browser.headers['content-type'], 'text/html; charset=utf-8')
```

To follow a link:

```
browser.getLink('Edit').click()
```

This gets a link by its text. To get a link by HTML id:

```
browser.getLink(id='edit-link').click()
```

To verify the current URL:

```
self.assertEqual(portalURL + '/edit', browser.url)
```

To set a form control value:

```
browser.getControl('Age').value = u"30"
```

This gets the control by its associated label. To get a control by its form variable name:

```
browser.getControl(name='age:int').value = u"30"
```

See the zope.testbrowser documentation for more details on how to select and manipulate various types of controls.

To submit a form by clicking a button:

```
browser.getControl('Save').click()
```

Again, this uses the label to find the control. To use the form variable name:

```
browser.getControl(name='form.button.Save').click()
```

To simulate HTTP BASIC authentication and remain logged in for all requests:

```
from plone.app.testing import TEST_USER_NAME, TEST_USER_PASSWORD

browser.addHeader('Authorization', 'Basic %s:%s' % (TEST_USER_NAME, TEST_USER_PASSWORD,))
```

To simulate logging in via the login form:

```
browser.open(portalURL + '/login_form')
browser.getControl(name='__ac_name').value = TEST_USER_NAME
browser.getControl(name='__ac_password').value = TEST_USER_PASSWORD
browser.getControl(name='submit').click()
```

To simulate logging out:

```
browser.open(portalURL + '/logout')
```

6.9.1 Debugging tips

By default, only HTTP error codes (e.g. 500 Server Side Error) are shown when an error occurs on the server. To see more details, set `handleErrors` to False:

```
browser.handleErrors = False
```

To inspect the error log and obtain a full traceback of the latest entry:

```
from Products.CMFCore.utils import getToolByName

errorLog = getToolByName(portal, 'error_log')
print errorLog.getLogEntries()[-1]['tb_text']
```

To save the current response to an HTML file:

```
open('/tmp/testbrowser.html', 'w').write(browser.contents)
```

You can now open this file and use tools like Firebug to inspect the structure of the page. You should remove the file afterwards.

CHAPTER 7

Comparison with ZopeTestCase/PloneTestCase

plone.testing and `plone.app.testing` have in part evolved from `ZopeTestCase`, which ships with Zope 2 in the `Testing` package, and Products.PloneTestCase, which ships with Plone and is used by Plone itself as well as numerous add-on products.

If you are familiar with `ZopeTestCase` and `PloneTestCase`, the concepts of these package should be familiar to you. However, there are some important differences to bear in mind.

- `plone.testing` and `plone.app.testing` are unburdened by the legacy support that `ZopeTestCase` and `PloneTestCase` have to include. This makes them smaller and easier to understand and maintain.
- Conversely, `plone.testing` only works with Python 2.6 and Zope 2.12 and later. `plone.app.testing` only works with Plone 4 and later. If you need to write tests that run against older versions of Plone, you'll need to use `PloneTestCase`.
- `ZopeTestCase`/`PloneTestCase` were written before layers were available as a setup mechanism. `plone.testing` is very layer-oriented.
- `PloneTestCase` provides a base class, also called `PloneTestCase`, which you must use, as it performs setup and tear-down. `plone.testing` moves shared state to layers and layer resources, and does not impose any particular base class for tests. This does sometimes mean a little more typing (e.g. `self.layer['portal']` vs. `self.portal`), but it makes it much easier to control and re-use test fixtures. It also makes your test code simpler and more explicit.
- `ZopeTestCase` has an `installProduct()` function and a corresponding `installPackage()` function. plone.testing has only an `installProduct()`, which can configure any kind of Zope 2 product (i.e. packages in the `Products.*` namespace, old-style products in a special `Products` folder, or packages in any namespace that have had their ZCML loaded and which include a `<five:registerPackage />` directive in their configuration). Note that you must pass a full dotted name to this function, even for "old-style" products in the `Products.*` namespace, e.g. `Products.LinguaPlone` instead of `LinguaPlone`.
- On setup, `PloneTestCase` will load Zope 2's default `site.zcml`. This in turn will load all ZCML for all packages in the `Products.*` namespace. `plone.testing` does not do this (and you are strongly encouraged from doing it yourself), because it is easy to accidentally include packages in your fixture that you didn't intend to be there (and which can actually change the fixture substantially). You should load your package's ZCML explicitly. See the plone.testing documentation for details.
- When using `PloneTestCase`, any package that has been loaded onto `sys.path` and which defines the `z3c.autoinclude.plugin:plone` entry point will be loaded via z3c.autoinclude's plugin mechanism. This loading is explicitly disabled, for the same reasons that the `Products.*` auto- loading is. You should load your packages' configuration explicitly.
- `PloneTestCase` sets up a basic fixture that has member folder enabled, and in which the test user's member folder is available as `self.folder`. The `plone_workflow` workflow is also installed as the default.

`plone.app.testing` takes a more minimalist approach. To create a test folder owned by the test user that is similar to `self.folder` in a `PloneTestCase`, you can do:

```
import unittest2 as unittest
from plone.app.testing import TEST_USER_ID, setRoles
from plone.app.testing import PLONE_INTEGRATION_TESTING

class MyTest(unitest.TestCase):

    layer = PLONE_INTEGRATION_TESTING

    def setUp(self):
        self.portal = self.layer['portal']

        setRoles(self.portal, TEST_USER_ID, ['Manager'])
        self.portal.invokeFactory('Folder', 'test-folder')
        setRoles(self.portal, TEST_USER_ID, ['Member'])

        self.folder = self.portal['test-folder']
```

You could of course do this type of setup in your own layer and expose it as a resource instead.

- To use zope.testbrowser with `PloneTestCase`, you should use its `FunctionalTestCase` as a base class, and then use the following pattern:

```
from Products.Five.testbrowser import Browser
browser = Browser()
```

The equivalent pattern in `plone.app.testing` is to use the `FunctionalTesting` test lifecycle layer (see example above), and then use:

```
from plone.testing.z2 import Browser
browser = Browser(self.layer['app'])
```

Also note that if you have made changes to the fixture prior to calling `browser.open()`, they will *not* be visible until you perform an explicit commit. See the `zope.testbrowser` examples above for details.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search