

Zope 3 Views

One of the nicest things that Zope 3 brought us is a way to manage view logic.



Martin Aspeli

Plone 2.5 brings us closer to the promised land of Zope 3. Zope 3 brings us a new way of working. This tutorial will show how to marry the old and the new, to make Plone products that are more extensible, better tested and easier to maintain.

Page 15 of 15.

In Zope 2, a view (be that a view of a content object, or a more standalone template) typically consists of a Zope Page Template that pulls in data from its context. The problem is that non-trivial templates usually require some kind of "view logic" or "display logic". People tend to put these in a few places:

- Complex *python:* expressions in the ZPT. This is bad because it makes your templates hard to understand, and because there is a limit to what you can do with one-line Python expressions.
- External Python Scripts in a skin layer that get acquired in the page template, e.g. [here/calculateDate](#). This is bad because it is cumbersome to create a new file for something which may be quite trivial, because all such scripts are part of a global namespace (and thus there may be conflicts between two different scripts with the same name), and also because Python scripts in the skin layers (and *python:* expressions) are slower than filesystem Python code and more restricted.
- A custom tool that provides some necessary functionality. This is bad because a tool is a singleton, so you will probably need to explicitly pass around a context. Tools are also part of that same global namespace (by way of acquisition from the portal root), and are a hassle to create and install.
- Methods on the context content object (where applicable). This is bad because it mixes presentation logic and the model (the schema) and storage logic. This often leads to an explosion of methods on each content type that are highly specific to a particular template. This pattern also requires that you have the ability to add new methods to the content type class, even if you are just adding a new view template for it.

As usual, these problems indicate a lack of separation of concerns. Zope 3's answer is a view - a class (typically) which may be associated with a template.

Views are multi-adapters

You will often hear that views are named multi-adapters of a context and a request. In fact, the concept of a multi-adapter originated in the need for views. For most practical purposes, you can forget about this - it is an implementation detail. However, you may sometimes need to look up views yourself, which can be done using:

```
from zope.component import getMultiAdapter
myView = getMultiAdapter((context, request), name='my_view')
```

More importantly, you need to know that to access the context the view is operating on inside that view, you can use *self.context*, and to access the request (including form variables submitted as part of that request, if applicable), using *self.request*.

Explicitly acquiring views

One of the easiest ways of using views with existing code is to make page templates in a skin layer as you normally would, and then acquire a view object that is used for rendering logic. One of the main reasons for using this approach is that it allows page templates to be customised using the normal skin layer mechanism. This approach is used extensively in Plone 2.5. Here's an example from the "recent" portlet, starting with *portlet_recent.pt*:

```
...
<tal:recentlist tal:define="view context/@@recent_view;
                        results view/results;">
    ...
    <tal:items tal:repeat="obj results">
        ...
    </tal:items>
    ...
</tal:recentlist>
```

The important line here is `context/@@recent_view`. This will look up a view named *recent_view* relative to the current context (*context* in page templates is a now-preferred alias for the *here* variable that was used before - *here* still works in Zope 2 templates, but is gone in Zope 3).

This view is defined by a class and a ZCML directive. The ZCML directive looks like this:

```
<browser:view
    for="*"
    name="recent_view"
    class=".portlets.recent.RecentPortlet"
    permission="zope.Public"
    allowed_attributes="results"
/>
```

Actually, this is not exactly what's in the file in Plone, since Plone is working around a few Zope 2.8 issues, but basically, this says that the view is available on all types of contexts (*for="*" - this could specify a dotted name to an interface if needed, more on that below*), has the name *recent_view*, is public (because of the magic permission *zope.Public*) and that when acquired, the attribute (method) *results* is allowed - more attributes could be specified separated by whitespace. The class that is referenced contains the view implementation. Here it is, again slightly modernised:

```

from Products.Five.browser import BrowserView
from Products.CMFCore.utils import getToolByName

from Acquisition import aq_inner

class RecentPortlet(BrowserView):
    """The recent portlet
    """

    def results(self):
        """Get the search results
        """
        context = aq_inner(self.context)
        putils = getToolByName(context, 'plone_utils')
        portal_catalog = getToolByName(context, 'portal_catalog')
        typesToShow = putils.getUserFriendlyTypes()
        return self.request.get(
            'items',
            portal_catalog.searchResults(sort_on='modified',
                                        portal_type=typesToShow,
                                        sort_order='reverse',
                                        sort_limit=5)[:5])

```

The use of `aq_inner()` on `self.context` is not strictly necessary always, but is a useful rule of thumb to make acquisition do what you expect it to do (this is because the *BrowserView* base class extends *Acquisition.Explicit*, which causes *self.context* to gain an acquisition wrapper that can mess with its acquisition chain).

Views with templates

Zope 3 does not use views in this way. Instead, you would bind the template to the browser view explicitly. The main drawback of this technique is that the template is not present in the *portal_skins* tool, and so cannot be customised through-the-web. This may be possible in future versions of Zope and CMF, but for now the full-blown view technique is best used when it is not necessary to customise views through-the-web. Of course, you can still override view registrations using ZCML on more specific interfaces or an *overrides.zcml*.

Here is a view for departments in the *charity* example product, under *charity/browser/configure.zcml*. Notice how this entire XML file is in the *browser* namespace, and thus it is unnecessary to prefix each directive with *browser*:

```

<configure xmlns="http://namespaces.zope.org/browser"
           i18n_domain="charity">

    <page
        name="charity_department_view"
        for="Products.borg.interfaces.IDepartmentContent"
        class=".department.DepartmentView"
        template="department.pt"
        permission="zope2.View"
    />

    ...

</configure>

```

Here, we explicitly state that this view is only available for *IDepartmentContent* objects. This means that if you try to invoke `@@charity_department_view` on anything that does not provide this interface, you will get a lookup error. The view is protected by the Zope 2 *View* permission. Also note that there is no *allowed_attributes* (or *allowed_interface*) attribute here. This is because the view is not intended to be used by other templates (if they tried, they would get an *Unauthorized* error when trying to access any attribute of the view) - all the logic is in the *department.pt* template.

The *department.pt* template is found in *charity/browser*, the same directory as the *configure.zcml* file above. You can use relative paths like *./templates/...* if necessary to point to the template file on the filesystem. Here is the class:

```

from Products.Five.browser import BrowserView
from Products.borg.interfaces import IDepartment

class DepartmentView(BrowserView):
    """A view of a charity department"""

    def __init__(self, context, request):
        self.context = context
        self.request = request

    def name(self):
        return self.context.Title()

    def managers(self):
        return self.context.getManagers()

    def details(self):
        return self.context.Description()

```

And here is the template that uses these methods:

```

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
lang="en"
metal:use-macro="here/main_template/macros/master"
i18n:domain="charity">
<body>
<metal:main fill-slot="main">

    <div metal:use-macro="here/document_actions/macros/document_actions">
        Document actions (print, sendto etc)
    </div>

    <h1 class="documentFirstHeading" tal:content="view/name" />

    <table class="listing vertical" style="float:right" tal:condition="view/managers">
        <tr>
            <th>Manager(s)</th>
            <td>
                <div tal:repeat="obj view/managers">
                    <a href="#" tal:attributes="href obj/absolute_url" tal:content="obj/Title" />
                </div>
            </td>
        </tr>
    </table>

    <div tal:content="structure view/details" />

    <metal:listing use-macro="here/folder_listing/macros/listing" />

    <div class="visualClear"><!--></div>
</metal:main>
</body>
</html>

```

Now, you can go to a hypothetical URL */mydept/@@charity_department_view* to see this view rendered. In fact, this is set as the *view* and (*Default*) aliases for the *Department* content type when *charity* is installed, so the user will never see this URL.

Views without templates

It is also possible to make views without templates. This is useful if you need a URL to submit that does some processing. That processing would normally be done in the `__call__()` method, as in the hypothetical example below:

```
<browser:view
  name="modify_customer"
  for=".interfaces.ICustomer"
  class=".customer.ModifyCustomerView"
  permission="cmf.ModifyPortalContent"
/>
```

Now, we could write a form that has `action="@@modify_customer"`, which would result in this being called:

```
class ModifyCustomerView(BrowserView):
    """Modify a customer from a form
    """

    def __call__(self):
        name = self.request.form.get('name', None)
        dog = self.request.form.get('dog', None)

        self.context.name = name
        self.context.dog = dog

        self.request.response.redirect('@@customer_view')
```

This is obviously a simplified example, but the important thing to realise is that the view will tend to use `self.context` and `self.request` to interact with the rest of the portal.

[Log in to add comments](#)