

# Views and viewlets for Plone 3.0 Product Development

Tom Lazar

# Who are you?

- *Not* a veteran Zope Developer
- A Plone integrator with Zope/Plone 2.x experience (CMF based skins)
- Entirely new to Plone (Congratulations!)
- Sceptical and / or confused about this stuff

# Who am I?

- Self-employed Developer, Sysad and Consultant, based in Berlin, Germany
- Started working with Plone in 2003
- Contributor since April 2006 (Archipelago Sprint, where development of 3.0 started)
- Started Plone 3.0 based product in March 2007, following trunk

# History I

## CMF skins

- A skin is a collection of templates and resources (CSS, JS, images, scripts, macros)
- CMF skins have *ordered layers*
- customization = copy to a higher layer
- Acquisition based lookup (“Which template is producing this markup?!”)

# History II

## Macros

- re-use sub-page templates (navigation, etc.)
- Acquisition based lookup (“Which macro is *really* being called?!”)
- Clumsy parameter handling

# History III

## Slots

- “Macros on steroids”
- i.e. add CSS links to header from body template, advanced loops
- extremely powerful
- only one macro per slot, though

# History III

## Python scripts

- security limitations (can't use `unrestrictedSearchResults`, can't import all packages etc.)
- cumbersome to debug (`enablesettrace`)
- Clumsy parameter handling (`pyflakes`)

# History III

## Python scripts

- No subclassing (let alone interfaces)
- Acquisition based lookup (“Which script is being called?!”)
- hard (impossible?) to test

# History IV

## The Good

- very straightforward: copy, edit, reload
- very flexible: easy to override defaults
- instant gratification (TTW support)
- large factor in Zope 2's success

# History V

## The Bad

- “implicit magic”: WTF is going on here?!
- tendency to overload templates with application logic (‘dummy’ calls, anyone?)
- “maze of python scripts”
- in summary: “messy”

# History VI

## The Ugly

“Acquisition is a  
jealous mistress”

Martin Aspeli

‘Nuff said!

# Enter Zope3

- Rewritten from the ground up
- “Component Architecture”: Interfaces, Adapters... and Views
- pluggable via configuration a.k.a. “XML sit-ups” (Grok is changing that, though)
- lots of other Goodies (buildout, eggs, events)

# Benefits of the CA

- makes it easy to write small, specific, easy to understand code
- re-usability
- explicit (it never guesses! no magic!)

# portalstatusmessages

## Interface

```
from zope.interface import Interface, Attribute

class IMessage(Interface):
    """A single status message."""

    message = Attribute('The text of this message. Usally a Message object.')

    type = Attribute('The type of this message.')

class IStatusMessage(Interface):
    """An adapter for the BrowserRequest to handle status messages."""

    def addStatusMessage(text, type=''):
        """Add a status message."""

    def showStatusMessages():
        """Removes all status messages and returns them for display.
        """
```

## Adapter

```
from base64 import encodestring, decodestring
from pickle import dumps, loads
import sys

from zope.annotation.interfaces import IAnnotations
from zope.i18n import translate
from zope.interface import implements

from Products.statusmessages import STATUSMESSAGEKEY
from Products.statusmessages.message import Message
from Products.statusmessages.interfaces import IStatusMessage

import logging
logger = logging.getLogger('statusmessages')

class StatusMessage(object):
    """Adapter for the BrowserRequest to handle status messages.

    Let's make sure that this implementation actually fulfills the
    'IStatusMessage' API.

    >>> from zope.interface.verify import verifyClass
    >>> verifyClass(IStatusMessage, StatusMessage)
    True
    """
    implements(IStatusMessage)

    def __init__(self, context):
        self.context = context # the context must be the request

    def addStatusMessage(self, text, type=''):
        """Add a status message.
        """
        text = translate(text, context=self.context)
        annotations = IAnnotations(self.context)

        old = annotations.get(STATUSMESSAGEKEY, self.context.cookies.get(STATUSMESSAGEKEY))
        value = _encodeCookieValue(text, type, old=old)
        self.context.RESPONSE.setCookie(STATUSMESSAGEKEY, value, path='/')
        annotations[STATUSMESSAGEKEY] = value

    def showStatusMessages(self):
        """Removes all status messages and returns them for display.
        """
        annotations = IAnnotations(self.context)
        value = annotations.get(STATUSMESSAGEKEY, self.context.cookies.get(STATUSMESSAGEKEY))
        if value is None:
            return []
        value = _decodeCookieValue(value)
        # clear the existing cookie entries
        self.context.cookies[STATUSMESSAGEKEY] = None
        self.context.RESPONSE.expireCookie(STATUSMESSAGEKEY, path='/')
        annotations[STATUSMESSAGEKEY] = None
        return value

    def _encodeCookieValue(text, type, old=None):
        """Encodes text and type to a list of Messages. If there is already some old
        existing list, add the new Message at the end but don't add duplicate
        messages.
        """
        results = []
        message = Message(text, type=type)

        if old is not None:
            results = _decodeCookieValue(old)
        if not message in results:
            results.append(message)
        # we have to remove any newlines or the cookie value will be invalid
        return encodestring(dumps(results)).replace('\n','')

    def _decodeCookieValue(string):
        """Decode a cookie value to a list of Messages.
        The value has to be a base64 encoded pickle of a list of Messages. If it
        contains anything else, it will be ignored for security reasons.
        """
        results = []
        # Return nothing if the cookie is marked as deleted
        if string == 'deleted':
            return results
        # Try to decode the cookie value
        try:
            values = loads(decodestring(string))
        except: # If there's anything unexpected in the string ignore it
            logger.log(logging.ERROR, '%s \n%s',
                'Unexpected value in statusmessages cookie',
                sys.exc_value
            )
        return []
        if isinstance(values, list): # simple security check
            for value in values:
                if isinstance(value, Message): # and another simple check
                    results.append(value)
        return results
```

# Views + Templates

Adapters  
(Application Logic)



View



Template

# Views + Templates

Let's dumb this down further ;-)

# Views + Templates

- Think “View = Dictionary”
- Views collect data from the application into a dictionary
- Templates take the data from the dictionary and insert it into the markup

# Views + Templates

- Templates insert, replace and loop (conditionally) – keep them ‘dumb’
- use methods (with `@property`, `@memoize` at your discretion)
- use kw dictionary for simple attributes:  
`kw.update('foo', bar)`

# Views + Templates

- Templates insert, replace and loop (conditionally) – keep them ‘dumb’
- use methods (with `@property`, `@memoize` at your discretion)
- use kw dictionary for simple attributes:  
`kw.update('foo', bar)`

# Views + Templates

Demo

# Views + Templates

- avoid using `self.foo` for attributes (for consistency's sake)
- python expressions in templates are not inherently evil! (only to abuse them is)
- python expressions are ca. 3x faster than path expressions!

# On to the practical stuff

Questions?

# A Simple View

```
from Products.Five.browser import BrowserView

class MyView(BrowserView):

    def __call__(self, *args, **kw):
        kw.update({'foo' : self.bar()})
        return super(MyView, self).__call__(*args, **kw)

    @property

    def some_value(self):
        return something_complicated()
```

# A Simple Viewlet

```
from Products.Five.browser import BrowserView

class MyView(BrowserView):

    def __call__(self, *args, **kw):
        kw.update({'foo' : self.bar()})
        return super(MyView, self).__call__(*args, **kw)

    @property

    def some_value(self):
        return something_complicated()
```

# Debugging Views

```
from Products.Five.browser import BrowserView

class MyView(BrowserView):

    def __call__(self, *args, **kw):

        import pdb ; pdb.set_trace()

        return super(MyView, self).__call__(*args, **kw)
```

# Context

```
from Acquisition import aq_inner
from Products.Five.browser import BrowserView
class MyView(BrowserView):
    def some_view_function(self):
        context = aq_inner(self.context)
```

“If you forget the `aq_inner()` it will probably still work 9 times out of 10, but the 10th time you're screwed and wondering why you're getting insane errors about user folders and attributes not being found.”

Martin Aspeli

# View Interfaces

- a contract between developer and designer

```
from zope.interface import Interface
from zope.interface import implements
from Products.Five.browser import BrowserView
```

```
class ISummaryPage(Interface):
    def summary():
        pass
    def body():
        pass
```

```
class IntroPageView(BrowserView):
    implements(ISummaryPage)

    def summary(self):
        return foo()
```

# Dos and Don'ts

- don't use `__init__()`
- use `__call__()`
- don't use `context/foo` (expensive!)
- use `kw + options/foo` and/or Interfaces

# Hack No. 1

## View + skin template

For extra flexibility you can hook up a Five view with a skin based template:

```
from Acquisition import aq_acquire
from Products.Five.browser import BrowserView

class IntroPageView(BrowserView):
    def __call__(self, *args, **kw):
        kw.update({'some_name' : nifty_method()})
        return aq_acquire(self.context, 'intro-page')(**kw)
```

# Skin switching

- Use a different skin depending on URL
- Useful for editing sites with heavily customized public skin
- For example: use Plone's default skin when accessed via HTTPS
- Old method with External Method and 'Set Access Rule' no longer works (plone.theme marks request with default skin before rule is processed)
- New method is actually easier and less code

# Skin switching

## The Code

```
def setskin(site, event):  
    if event.request.URL.startswith('https'):  
        site.changeSkin("Plone Default", event.request)
```

## The Glue

```
<subscriber  
    for="Products.CMFP1one.interfaces.IPloneSiteRoot  
        zope.app.publication.interfaces.IBeforeTraverseEvent"  
    handler=".skinswitcher.setskin" />
```

# Caching

Two basic approaches:

- cache results of view methods: memoize
- cache entire view/viewlet: lovely.viewcache

# Caching

```
from plone.memoize.instance import memoize
from Products.Five.browser import BrowserView
class MyView(BrowserView):
    @memoize
    def expensiveMethod(self):
        return foo
```

# lovely.viewcache

```
from lovely.viewcache.view import  
cachedView
```

```
class View(BrowserView):  
    pass
```

```
CachedView = cachedView(View)
```

Now use `CachedView` instead of `View` in your `.zcml`  
Works just the same for `Viewlets`

# lovely.viewcache

- Entire page usually not cacheable
- But caching viewlets is particularly useful
- By using views and viewlets you futureproof your product for later scalability!

# subclass `BrowserView`

- common methods for all your views
- group and share similar behaviour
- hierarchy easier to debug than Acquisition!

# Hack No. 2: custom viewlet per page

Usecase: custom logo viewlet for frontpage only.  
Didn't want to add extra logic to view class and  
template just for that.

Solution: custom *template* for frontpage

```
<browser:viewlet
  name="plone.logo"
  manager="plone.app.layout.viewlets.interfaces.IPortalHeader"
  class=".viewlets.LogoViewlet"
  permission="zope2.View"
  layer=".interfaces.IMySkinSpecific"
/>
```

# Hack No. 2: custom viewlet per page

```
from Products.CMFPlone.interfaces import IPloneSiteRoot
from plone.app.layout.viewlets.common import LogoViewlet as
LogoViewletBase
class LogoViewlet(LogoViewletBase):

    _template = ViewPageTemplateFile('logo.pt')
    _frontpage_template = ViewPageTemplateFile('logo-frontpage.pt')

    def render(self):
        if IPloneSiteRoot.providedBy(self.context):
            return self._frontpage_template()
        else:
            return self._template()
```

# Hack No. 2: custom viewlet per page

That's stupid, don't do that!

Viewlet accepts view attribute, problem solved!

```
<browser:viewlet
  name="plone.logo"
  manager="plone.app.layout.viewlets.interfaces.IPortalHeader"
  class=".viewlets.FrontpageLogoViewlet"
  permission="zope2.View"
  layer=".interfaces.IMySkinSpecific"
  view="IFrontpageView"
/>
```

# Fun with Views

- Remember: Views are just multi-adapters that return (in most cases) HTML
- ZPT is just a default behavior
- Let's use... Genshi

# Genshi

```
from zope.publisher.browser import BrowserPage

class HelloGenshi(BrowserPage):

    def __call__(self):

        tmpl = loader.load('helloworld.html')

        stream = tmpl.generate(

            request=self.request,

            who=self.request.principal.id

        )

        return stream.render('html',

                             doctype='html')
```

# Genshi

```
<div xmlns:py="http://genshi.edgewall.org/">
  <p>Hello $who</p>
  <p>Hello ${request.principal.id}</p>
  <table>
    <tr py:for="key in request"
        py:if="key.startswith( 'HTTP' )">
      <td>$key</td>
      <td>${request[key]}</td>
    </tr>
  </table>
</div>
```

# Genshi

- everything you love about ZPT
- (hardly) anything you hate about ZPT
- also works with non-markup
- still valid XHTML
- less verbose

Thank you.