Best Practices for Development

North American Symposium II

Author: Joel Burton <joel@joelburton.com>
Copyright: Copyright 2006 Joel Burton
Covering: Plone 2.1 and newer

Contents

1 Best Practices for Plone Development

1.1 Introduction

1.1.1 Handouts and Examples

- http://temp.joelburton.com/symposium

1.1.2 Why Do We Need This?

- Our inspiration:

```
Between the idea
And the reality
Between the motion
And the act
Falls the Shadow

  T.S. Eliot, The Hollow Men
```

1.1.3 Safety

- Constancy and Fortitude

1.1.4 Documentation

- Truth and Unity

1.1.5 Re-Usability

- Mercy and Clemency

1.1.6 Streamlined Development

- Virtuous Example

1.2 Products

1.2.1 Concept

- Build products for each major feature

  For every feature of the site that might be re-used in another site, build a product for this. For example, I recently wanted a Plone site that allowed Plone to use the "sent to friend" feature for any page, based on URL, not just based on Plone content. This only required changing 3 skins from CMFPlone, and could have been customized for this site. Instead, I chose to separate it out to the product SendToURL, allowing me to add this to any site instantly. And, if I want to customize these skins further for another site, I can do so.

- Build a site product for site customization

  For the site itself, build a "site product" to hold, at the very least, the setup scripts, graphics, skins, etc. Keep this information, which is very site-specific, from the features.

- Avoid disconnected ExternalMethods, scripts, etc.

  ExternalMethods or other kinds of page templates or scripts that are hanging out in your ZODB end up being disconnected and hard to maintain. Make them part of your feature products, or if they're only for the site itself, part of your site product.

1.2.2 Products

- Each product contains
  - Classes of content types
  - Skin folders
  - Installation scripts

1.2.3 Framework

- Ordinary setup:

```
__init__.py
config.py
content/
  __init__.py
  YourClass.py
skins/
  productname/
    class_view.pt
    class_edit.py
Extensions/
  Install.py
  YourExternalMethod.py
doc/
  YourDocumentation.txt
VERSION.txt
```

1.2.4 Framework: config.py

- Anything being shared among other scripts
  - Common import area

1.2.5 Framework: content/

- Contains all new content types
  - Usually Archetypes-based
- `__init__.py` here just loads types

Many existing products use the directory name 'types' for this, but this has a conflict with the python built-in types.

1.2.6 Framework: skins/

- Contains the skin folders
  - Not the skins directly
- Skin folder convention is lower-cased, no-space version of product name

---

1.2.7 Framework: Extensions

- Install.py script install product
  - Detailed later
- External methods for this product
  - Works same as site-wide external methods (in instance `Extensions`)
  - Except "Module name" includes Product
    - eg, `YourProduct.ExternalMethodFile`

1.2.8 Framework: doc/

- Documentation directory
- Not used by Plone

1.2.9 Framework: VERSION.txt

- Should contain just version number ("1.0.1")
- Shown in Plone control panel
- Easy way to tell what version is installed

1.2.10 Framework: __init__.py

- Registers product
- Registers skin folders
- Calls content/__init__.py (which registers types)

1.2.11 Site Product Overview

- Conceptual container for site customization/skins
  - eg, PressRelease product is standard
  - On parks site, different appearance is in custom skins
- Generallly just contains skins, not types
  - Unless simple and site-specific
- Make the highest skin path (other than custom)

1.2.12 ArchGenXML

- Turns UML data to AT classes
  - Quick RAD system for building Archetypes
  - Can use even without knowing UML
  - Can round-trip
- Creates entire package
  - Installers
  - Initializers
  - Classes

1.2.13 UML

- Good open source Java-based UML editors
  - ArgoUML is entirely Open Source
  - PoseidenUML has a community edition that is free
- O'Reilly's `Learning UML` is a good introduction

1.2.14 Converting UML

- Save diagram to compatible format
  - XMI, .zargo, .zuml
- Run `ArchGenXML.py` over it:

```
$ python ArchGenXML.py your.xmi ProductName
```

- Use experimental web server
  - http://uml.joelburton.com
  - Only accepts XMI files

1.2.15 ArchGenXML Tips

- Don't try to set field types, required, etc.
  - Unless you like UML and the editor
  - Easy to do this stuff in Python
- Don't try to use UML diagram from one program in another
  - It doesn't always work

1.2.16 Skeletor

- Skeletor[3]: "Product skeleton builder"
- Pluggable, so gurus can write new builders
- Not quite fully baked
- Keep your eye on this one!

1.3 Filesystem Skins

1.3.1 Through the Web Editing

- `TEXTAREA` boxes suck
- ExternalEditor makes it suck less
  - ExternalEditor is best for content editing

  Don't underestimate the value of this. I use vim, which launches quickly (so it can be used easily with ExternalEditor TTW), and which practically every server already has installed (so I can use through ssh). However, even considering that, I find myself almost twice as productive being able to work in my exact editing environment, with my macros, scripts, paths, etc., already set.

1.3.2 More Web Editing Problems

- No version control
  - What did I do?
  - Why did I do this?
  - *Especially* for HTML/JS/CSS
- No grep, tags, etc.

  grep and find are the godsends of working on the filesystem. Once you've reacquainted yourself with them after years of ignoring them for the ZMI, you'll wonder what you were thinking.
  - Refactoring extra-painful

1.3.3 File-System Stored Skins

- File itself contains body
- `.metadata` file contains everything else
  - Security
  - Title
  - FormController information

  These are the accompany files that hold all the "other stuff", titles, security settings, proxy roles for PythonScripts, and more. These replace the .properties files in earlier versions of the CMF.

  One common mistake is to customize a skin object file (say, 'foo.py') by copying it to a new directory, but not copying any 'foo.py.metadata'. Note that the .metadata file **must** be in the same directory, so in this case, the customized foo object is used, and it does not get the settings in its metadata file. If this contained important things, like security or proxy settings, this could be disastrous.

  In CMFPlone, there are examples of all of the types that can be stored on the filesystem including feature like proxy settings, titles, security settings. 'grep' is your friend here: 'grep -r --include="*.metadata" proxy *' Will give you an example of using a proxy setting in .metadafiles.

  Inline emphasis start-string without end-string.

  Inline emphasis start-string without end-string.

1.3.4 Filenames Matter

- Page Templates: `*.pt`
- PythonScripts: `*.py`
- Images: `*.gif`, `*.png`, etc.
- DTML Methods: `*.dtml`
- Extensions are **not** part of object Id, though
  - So, `foo.css.dtml` gets ID `foo.css`
  - Image extensions are kept

1.3.5 Sample foo_view.pt.metadata

- Not much:

```
[default]
title=Register a User
```

- Not required, either

1.3.6 Sample foo_edit.py

- Binding info in top comments:

```
## Python Script "foo_edit"
##bind container=container
##bind context=context
##bind namespace=
##bind script=script
##bind state=state
##bind subpath=traverse_subpath
##parameters=new_body, attribA

context.doStuff(new_body, attribA)
return context()
```

1.3.7 Complex foo_view.pt.metadata

- Additional things:

```
[default]
title=Register a User
proxy=Manager,Anonymous

[validators]
validators = validate_registration

[actions]
action.failure=traverse_to:string:join_form
action.success=traverse_to:string:registered
action.prefs=traverse_to:string:prefs_users_overview
```

- Not required, either

1.3.8 ZSQL Methods

- DTML comment holds SQL meta-stuff:

```
<dtml-comment>
  connection_id: my_conn
  arguments: fname lname
</dtml-comment>

SQL STATEMENT ...
```

There's no example of a working ZSQLMethod shipped with CMFPlone, CMFDefault, or CMFPlone. For a discussion of ZSQLMethods and an example of a filesystem-stored ZSQLMethod, see http://plone.org/Members/pupq/reldb.

1.3.9 Skin Changes

- Only visible during DEBUG mode
  - Set in `zope.conf`
  - Leave on for development!
- Otherwise must restart Zope
  - Or refresh product in Control Panel

  Refreshing products saves the time for Zope to restart, but there can be minor side-effects that require a restart, anyway, depending on what the product does.

  There's no harm try this and seeing it works, though. To do so, drop a file in your product's top directory called `refresh.txt`, then you can refresh the product and see if this works.

  Note that if you're using SpeedPack under debug mode, you'll see changes to skin objects, as long as they don't get copied from one directory to another. However, if you customize a skin object to a new directory and are running SpeedPack under debug mode, Zope has already cached the old location of the skin object, and won't use the new location version. Restart or use product refresh to have Zope notice this.

1.3.10 Placeful Skin Objects

- Goal: no skins or scripts in ZODB. Only config results and content.

That is, our end goal is that the only thing in the ZODB will be our actual contentish objects and configuration settings made by scripts. All of our skins and all of our scripts will be on the filesystem.

- What about "placeful skin objects"?
  - Put stub object in ZODB
  - Call skin object

Sometimes, it's very helpful to have scripts or skins be placeful. For example, you might have a different logo in different parts of your site, and a common Zope practice for this would be to have 'logo.jpg' in the root of your site, and a different one in '/about'. However, this leaves a piece of skin in the site, and ruins our ability to maintain it well. Better is to have the root folder and '/about' folder have a property--say, logo_name, which tells us what logo to use in this area. Then, we can keep all of these logos on the filesystem, and have achieved our goal of just keeping the configuration part in the ZODB.

1.3.11 New FS-Stored Types

- Can create your own FS-stored types

  If there are non-contentish objects you use often, it's worthwhile creating the small script that will allow it to be stored on the filesystem. You can look at the code in CMFCore for storing the existing types (PageTemplates, PythonScripts, Files, etc.)

- 'FSExternalMethod'

  An example of taking an existing Zope object type and creating a FS-stored capable version of it. Taken from FileSystemSite.
  - Useful for your Zope add-ons

1.3.12 Atonement for Your Sins

- FSDump

  Sometimes, it's not possible to work on the filesystem. You may only have access to a web browser or have created many existing skin objects in the ZODB. FSDump will take existing skin objects and dump them out to the filesystem.

  - Dumps existing ZODB stuff to FS
  - Need to write dumpers for your FS-Stored types.

    If you've written custom FS-stored types, you'll have to write your own Dump plug-in for this. This is quite easy to do--see the code in FSDump for examples of how it dumps the basic types.
  - At the very least--FSDump for backups

1.3.13 Allowing Modules Imports

- Documentation in $ZOPE/lib/python/Products/PythonScripts
- Example, in product `__init__.py`:

```
from AccessControl import allow_module, allow_class

allow_module('zipfile')
from zipfile import ZipFile
allow_class(ZipFile)
```

1.4 Form Controller

1.4.1 CMFFormController

- New form handling/dispatching system
- Provides "controller" of MVC paradigm
  - Validation
  - Dispatching
- More flexible than simple marshalling

1.4.2 Dispatching / Actions

- **Template/Script**
  - Which script/template this affects
  - eg "document_edit_form"
- **Status**
  - What status this affects
  - "success", "failure"
  - Can emit your own

1.4.3 Dispatching / Actions (2)

- **Context**
  - What kind of object/context this work on
  - eg “Document”, “News Item”
- **Button**
  - What button name was pushed
  - Allows “Ok”, “Cancel” buttons

1.4.4 Dispatching / Actions (3)

- **Action**
  - What happens next
- **Argument**
  - Argument for action

1.4.5 Dispatching / Actions (4)

- **Actions**
  - **redirect_to**: HTTP redirect to argument
    - eg “http://yahoo.com”
  - **redirect_to_action**: HTTP redirect to action
    - eg “view”, “edit”
  - **traverse_to**, **traverse_to_action**: same, but no redirect, just calls
    - Keeps request object

1.4.6 Dispatching / Actions (5)

- **Arguments**
  - TALES expression
  - "string:http://www.yahoo.com"
  - "string:view"
  - or "python:", or just path, of course

1.4.7 Example Dispatching

- Form `friend_edit_form`, should go to `friend_edit`
  - Template: "friend_edit_form"
  - Status: "success"
  - Context: "Any" *or* "Friend"
    A subtle difference. It's possible that our `friend_edit_form` might be used for just `Friend` portal types. It's also possible that we re-use it for `CoWorker` portal types. If we want the behavior to be the same, we can keep this as "Any". If we want to have different dispatching for the two (perhaps coworker-editing needs a special `coworker_edit` script that does things differently, we can specify "Friend" here and put in a different action for "CoWorker".
  - Button: Any (unless you have cancel!)
  - Action: "traverse_to"
  - Argument: "string:friend_edit"

1.4.8 Controlled Scripts

- After validation what dispatching can go to
- Can use regular script
  - Controlled can participate in dispatching

1.4.9 Example Controlled Script

- `foo_edit` Controlled Script:

```
try:
  context.n = context.REQUEST.n
except:
  state.set(new_status='failure')
else:
  state.set(
      portal_status_message='n=' + n)
return state
```

1.4.10 Example Controlled Script: Notice

- We get *n* from `context.REQUEST`
  - Could also get with script parameters
- Set status message as before
- `return state` **required**

1.4.11 Dispatcher for Our Script

- Script `foo_edit` should go to "view" action:

```
- Template: "foo_edit
```

  - Status: "success"
  - Context: "Any" *or* "Foo"
  - Button: Any
  - Action: "redirect_to_view"
  - Argument: "string:view"
    Don't forget the `string:`!

1.4.12 CMFFormController Docs

- Included in package
  - Excellent
  - Examples here borrowed from there

1.4.13 Storing Definitions

- Stored on forms/scripts
- Stored on filesystem for FS skins/scripts
- Stored in portal_formcontroller tool itself
- Set in request object

1.4.14 Storing Actions/Validations on FS

- Stored in `.metadata` file for object:

```
[validators]
validators.[Type].[Button] = val1, val2

validators.Event.Save = validate_event
validators..Save = validate_event
validators = validate_id, validate_event
```

1.4.15 Stored on Tool

- Useful for overriding standard/product skins/scripts
  - Without having to customize skin/script

1.4.16 FormController Types

- Controller Page Template: `*.cpt`
- Validators: `*.vpy`
- Controlled Scripts: `*.cpy`
- `.metadata` files can contain Form Controller actions/validator info
  - As documented in FormController section

1.5 Skinning Plone

1.5.1 Possibilities for Skinning

Change only base_properties Easiest, still looks like Plone

Change CSS Requires CSS skills, not Plone-specific

Integrate HTML with main_template Complex, flexible

Separate CMS from retail view Can be easy, can be fastest

1.5.2 base_properties Site

1.5.3 CSS Skinning Site

1.5.4 Integration Site: Public

1.5.5 Integration Site: Private

1.5.6 Replacement Site: Public

1.5.7 Replacement Site: Private

1.5.8 CSS Skinning Site: Values

- Easy to maintain
- Benefit from Plone's accessibility
- Requires some design flexibility
  - Deep knowledge of CSS may be needed
  - Firefox `WebDeveloperExtensions` is your friend!

1.5.9 Skinning Practice: Integration

- Takes longer
  - Results in complex mix of Plone-isms and client HTML
- Keeps excellent editor/user experience
  - No sudden shifts in site
- Slower for viewing than replaced
  - Standard Plone UI is slow

1.5.10 Skinning Practice: Replacement

- Easy to accomplish
- Can be hard for users to understand
  - Best for small # of content editors
- Requires inclusion of global_defines material

1.5.11 Useful Plone CSS Items

- **#portal-top**: logo, user bar, top tabs
- **#portal-globalnav**: top tabs
- **#portal-personaltools**: user bar
- **#portal-breadcrumbs**
- **#portal-column-one, -two**: left, right slots
- **#content**: middle of O wrap
  - **.documentContent**: inner of content (no margins)

1.5.12 Useful Plone CSS Styles

- **#portal-footer**
- **#portal-colophon**
- **.portalMessage**: orange announcements
- **.documentByline**
- **table.listing**
- **.discreet**: quiet text (gray, small)

1.6 Version Control

1.6.1 Source Code Control

- Critical for teams

  If your team is larger than one, source code management is essential. It will allow your team to work together more quickly, with much less "stay out of this; I'm working on it", and much less "I'm not sure what this person was doing here". Hands-down, the successful implementation of SCM will be the biggest win for your team's coordination and performance.

- Helpful for code archaeology

  When faced with old code of your own, or someone else's code, seeing the log messages and way it was built is often incredibly helpful in finding bugs and maintenance.

- Branches

  Often, you'll work for a day or two on a new idea, only to figure out that it isn't working out, you've screwed up, and you can't remember all the billions of things you changed while on a tear to try out this new idea. This is exactly what branches in a version control system are meant to manage.

  Learn how to use branches for your VC system. They're easier that even in Subversion.

1.6.2 Version Control

- Subversion recommended
  - Similar to CVS, but redesigned from ground-up
  - Commits are for an entire changeset, not just files
  - Easier to understand relationship of commits
  - Sane API for utilities
- Zope isn't involved at all w/version control

1.6.3 Subversion in 3 Minutes: Starting

- `svn co http://user:pass@repos/product`
  - Sometimes, this may be `svn://` or `https://`
  - Creates directory of checked-out files

1.6.4 Subversion in 3 Minutes: Updating

- `svn up [files]`
  - Update this directory to repository
  - If `[files]` not given, do all
  - Do this often!

1.6.5 Subversion in 3 Minutes: Check-In

- `svn ci [files]`
  - Checks in changes
  - If `[files]` not given, do all
  - Calls up editor to enter comments

1.6.6 Subversion in 3 Minutes: History

- `svn log [files]`
  - Shows history of changes
  - Normally specify one or two files

1.6.7 Subversion Under Windows

- TortoiseSVN makes this a snap
  - Easy enough for non-technical users

  At first, the task of having non-developers use your version control system seems daunting. I've found, however, the non-developers can love it when they realize they can easily work off files on their computer and sync it with the server, *and* they understand that they don't have to "worry" about mistakes. It's all in how you sell this idea.

  - SCM is not just for coders!

1.6.8 Learning More About Subversion

- Understanding differences between files
- Merging and resolving conflicts
- Starting new branches and settings tags

All of these are covered in the excellent, free Subversion Book[2].

1.6.9 Tips for Version Control

- Log messages are your friend.
  - So don't treat them like your enemy.
- Refer to collector items in messages.

  It's good to pick a simple, standard syntax for this. I frequently use the phrase "Collector #123", and can build web tools that allow you to jump right to that bug to see the details of what you were tryin to fix. This helps close the loop on why you were making these changes in the first place.
  - Focus on why, not what.

    Of course you were editing 'login_form'. We can see that. Why, though, did you make those changes? What was broken? What client request does this address? This is the information you'll want later. Explanations of what you're doing should be in the code as comments, anyway.
  - Check in "pristine" copy as first copy.

    If you want to customize Plone's 'login_form', for example, don't copy it to your site product directory and immediately start hacking on it. Instead, copy it to your directory, *check it in right then*, in its pristine form, then start hacking away. Now, you've solved two problems: (a) it's trivial for you to diff revision 1 and revision 2 of this file to find out *why* you were customizing it in the first place, and (b) when Plone is upgraded and there are changes to the shipped 'login_form', it's much easier to incorporate those, since you know exactly how 'login_form' looked when you started, without having to dig around and find that version.

    This takes only a tiny bit of discipline and really pays off.
    * Easier to understand why you customized a CMFPlone skin.
- Keep a `CHANGES.txt` file for explaining larger changes in context
- Always check in working code
  - Or put it on a branch

1.7 Documentation

1.7.1 Documentation Ideas

- Consider using interfaces
- Definitely use your docstrings
- Alpha-test your documentation

1.7.2 Documentation Products

- DCWorkflowGraph
- DCWorkflowDump
- EpyDoc
- ArchGenXML

1.8 Setup and Scaffolding

1.8.1 ZODB Dread

- You know the feeling:
  - How the hell am I going to get this all back?

Everyone that's worked with Zope for more than a few months has encountered "ZODB Dread": that awful, sinking feeling that you've sunk a chunk of your very life into a single, binary-format object database, with no hope you'll ever be able to remember all the scripts, skins, properties, and settings you've put into it. You konw that if this puppy ever gets badly corrupted, you're going to be in a world of hurt.
This is what we want to avoid.

1.8.2 Throw Out Your ZODB

- *Throw out your ZODB. It's liberating.* - Kapil Thangavelu, (hazmat on #plone) ObjectRealms

---

1.8.3 Setup Scripts

- Useful way to keep track of config info on disk
  - Make calls via API
- New `setup` directory in product:

```
setup/
  __init__.py
  CustomSetup.py
```

1.8.4 Setup Script: __init__.py

- Boilerplate:

```
from Products.CMFPlone import MigrationTool
from Products.MyProduct.setup.
  CustomSetup import CustomSetup
MigrationTool.registerSetupWidget(CustomSetup)
```

1.8.5 Setup Script: CustomSetup.py

- Imports and function:

```
from Products.CMFPlone.setup.SetupBase \
    import SetupWidget
from zLOG import INFO
from Products.Archetypes.utils \
    import OrderedDict

def Americanize(self, portal):
  sprops=portal.portal_properties.site_properties
  sprops._updateProperty(
    'localTimeFormat','%B %e, %Y')
  return "Set American dates."

functions = OrderedDict()
functions['Americanize'] = Americanize
```

1.8.6 Setup Script: CustomSetup.py

- Boilerplate:

```
class CustomSetup(SetupWidget):
    type = 'MyProduct Setup'
    description="Setup for ..."

    def available(self): return functions.keys()
    def installed(self): return []

    def addItems(self, fns):
      out = []
      for fn in fns:
        out.append((functions[fn]
          (self, self.portal),INFO))
        out.append(
          ('Function %s applied' % fn, INFO))
      return out
```

1.8.7 Calling Setup

- `portal_migrations`, Setup
- Can extend to list only un-applied setup, etc.
  - Helpful when doing laptop/dev server sync
- Better: create a "customization policy" that spawns fully-customized site
  - Samplex product provides example of customization policy

1.8.8 GenericSetup

- (Was CMFSetup)
- Framework for inspecting ZMI tools and writing state
- Can import state to move back to snapshot
- Not quite fully-baked
  - Not all Plone tools are supported
  - No easy way to perform some config changes
  - Can't easily script creation of sample content
- Keep a watch on this!

1.9 Debugging

1.9.1 Error Log

- Location for details on error messages
- Accessible through Plone interface (also in ZMI)
- While developing, remove `Unauthorized` and `NotFound` filters

1.9.2 Output Debugging

- `print "foo=%s" % value`
- `plone_log(summary, text)`
- `raise "foo!", values`
  - Shows & stores REQUEST obj
  - Can make conditional on someting in request

1.9.3 Limitations of Output Debugging

- Output statements in many places
  - Have to go back and clean up later?
- Few “aha!” moments

1.9.4 Dropping to PDB Manually

- Run Zope in foreground mode
  - `zopectl fg`
- Drop into pdb at right place:

```
import pdb
pdb.set_trace()
```

1.9.5 PDB in 3 Minutes

- **h**: help
- **s**: step, **n**: next
- **r**: return, **c**: continue
- **w**: frames

1.9.6 PDB in 3 Minutes (2)

- **l**: list
- **p**: print, **pp**: pretty-print, **whatis**: what-is-object
- **b**: breakpoint:

```
b [file]:lineno | [obj.]function [, condition]
```

1.9.7 Disadvantages of PDB

- Same process as main Zope
  - You're holding it up!
- Can't get to pdb from PythonScripts
  - But can change this easily
- Must restart for code changes in products
- Not easy to use with Windows

1.9.8 ZEO Overview

- ZEO
  - Object server (ORB)
  - Zope server is ZEO client
  - Allows multiple Zope clients
- Use ZEO all the time

1.9.9 Creating a ZEO Server

- Script to create new ZEO server:

```
$SOFTWARE_HOME/bin/mkzeoinstance.py home [port]
```

  - *port* is port ZEO server runs on
- Edit `$INSTANCE_HOME/etc/zope.conf`
  - Comment out first zope_db main; uncomment second
  - Port should be set to ZEO server
  - Move existing `Data.fs` from `var` to `zeo/var`

1.9.10 Running a ZEO Server

- `zeoctl start`
- `zopectl start` (or `zopectl fg`)
- Don't have to restart ZEO for Zope changes

1.9.11 ZODB Shell Basics

- `zopectl debug`
- `app` is the root of the ZODB
  - Can walk to any object
  - No security checks or restrictions

1.9.12 ZODB Shell Examples

- For example:

```
app.plone.objectIds()
app.plone.portal_catalog(Title="Hello")
app.plone.Members.joel.doc.attrib=2
```

- Inspection: `dir()`
  - Need to control it's size

1.9.13 Transactions

- Automatically discarded at end of session
- Discard now:

```
get_transaction().abort()
```

- Commit now:

```
get_transaction().commit()
```

1.9.14 Synchronizing Transaction

- Sync your transaction to the most up-to-date:

```
app._p_jar.sync()
```

1.9.15 Limitations of ZODB Shell

- Not a real request
  - Missing REQUEST, RESPONSE, user, etc.
- Not really a user

1.9.16 Testing a Request

- Real request through ZPublisher:

```
import Zope
Zope.debug('/path/to/object')
```

- Returns output
- Happens in real, separate transaction

1.9.17 Testing a Request Options

- **d=1**: drop into pdb and debug request
- **pm=1**: (postmortem) drop into pdb on failure
- **t=1**: output timing
- **u='username:pass'**: authenticate request as user
- **extra={'key':'value'...}**: add to REQUEST object
- **p='file.prof')**: creating Python profile data

1.9.18 Example of Testing Request

- Debug Membership Tool's listMembers method:

```
pdb> from Products.CMFPlone.MembershipTool
       import MembershipTool

pdb> b MembershipTool.listMembers

Breakpoint ... /CMFPlone/MembershipTool.py:247

pdb> c

> .../MembershipTool.py(247)listMembers()
```

1.9.19 WingIDE Overview

- Commercial Python IDE/debugger
- Nice editor/IDE
  - But I don't use it for that
- Free if you only use to develop open source
  - But definitely worth paying for
- Personal version is probably fine

1.9.20 Setting Up WingIDE

- Install WingIDE (Windows, MacOS, Linux versions)
  - Does not have to be same machine as Zope server
- Install WingDBG[4] product on Zope server
  - Patching CMFCore and Zope no longer needed

1.9.21 Starting up WingIDE

- Enable Passive Listening
  - Preferences > Debugger > External/Remote
  - Only have to do once
- Optionally create project for site
  - Easier to navigate among files

1.9.22 Setup in ZMI for WingIDE

- Visit Control Panel
- Turn on debugger and connect to IDE
- Visit `http://localhost:50080/plone` and login
- All traffic through 50080 is monitored by Wing

1.9.23 Sample of Using Wing

- Open `Products/CMFCore/CatalogTool`
- Add breakpoint to `CatalogTool.searchResults`
- In browser for port 50080, search site

1.9.24 Useful Wing Features

- **Stack data**: interactive stack variables
- **Debug I/O**: Console output
- **Debug probe**: interactive Zope shell
- **Python shell**: normal Python shell
- **Watch**: Variables to watch

1.9.25 Debugging File-System Python Scripts

- Only system that supports feature
- Works the same as product code debug

1.9.26 Other Debugging Systems

- Monitor server
  - Outdated by ZEO
  - Documented in `$SOFTWARE_HOME/docs/DEBUGGING.txt`
- ActiveState Komodo
  - Excellent Python debugger/IDE
  - Poor Zope support

1.9.27 Other Debugging Systems II

- Boa Constructor
  - Open source IDE/debugger
  - Can be a tricky install
  - Interesting featureset but less stable/mature

1.10 Testing

1.10.1 Selenium Overview

- In-browser, functional testing
- Tests are straightforward to write
  - Even “client-safe”
- Running tests is esay and graphical
- Uses JavaScript

1.10.2 Selenium Concepts

- Actions
- Checks
- Element Locators

1.10.3 Selenium Locators

- Identifier (id then name)
- DOM
- XPath
- Link Text
  - requires prefix

1.10.4 Selenium Actions

- `open`
- `click`
- `type`
- `select`
- And more
  - Including "...AndWait" versions

1.10.5 Selenium Checks

- `assert` vs. `verify`
  - Assert stops the test on failure
- `verifyLocation`
  - Are we at the right URL
- `verifyTitle`
- `verifyElementPresent`

1.10.6 Selenium Form Checks

- `verifyValue`
  - Checks value of INPUT
- `verifySelected`
  - Checks value of SELECT

1.10.7 Selenium Text Checks

- `verifyText`
  - For a particular node, exact match
- `verifyTextPresent`
  - Anywhere on page
- `verifyTextNotPresent`

1.11 Developing With Others

1.11.1 You Have a Laptop: Use It

- It's not a $2000 ssh client
- Faster edit/test/debug cycle.
- No wires, no wireless
- Easier to work privately

1.11.2 Subversion for Sharing

- Work on your laptop
- When at the "right point", check-in your code
- 'rsync' to your sandbox on server
- Emergency changes on server can be handled, too

1.11.3 Simple Sandboxes

- Each developer gets a skin folder in your product
- rsync your skin changes to your skin folder
- Each developer has their own skinpath, with their skinfolder as most customized
- Developers can switch from their sandbox skinpath to common skinpath.

1.11.4 Snychronizing of Content

- Create starter content in setup scripts
- Create starter content in '.zexp' files
  - Can't change, though
- Use 'ZSyncer' to sync from one server to another
- Use AT 'XMLTool' or 'Marshall' to export/import XML of content
  - GenericSetup will help here in the future

1.11.5 More Information

- Examine good products
  - PloneHelpCenter (straightforward, AT product)
  - SimpleBlog (AT product with TTP prefs, etc)
  - Samplex (scaffolding, customization policies)
  - listen (uses of Five/z3 technology)
- My Site[1] tutorial
- `RichDocument` tutorial

1.11.6 CREDITS.txt

- Kapil Thangavelu
  - Inspiration to throw out my database and much more
- Rob Miller
  - Excellent Samplex product
- Ben Saller
  - Many excellent hints
- Alec Mitchell
  - listen for examples on Five

2 Footnotes

[1] http://www.neuroinf.de/PloneDevTutorial

[2] http://svnbook.red-bean.com/

[3] http://plone.org/products/skeletor

[4] http://wingware.com/downloads/wingide/2.1.0/zope