

- Zope 3 - Component Architecture with a Twist

Stephan Richter

1. Introduction to Zope
2. Goals and Design of Zope 2
3. Problems with Zope 2
4. Zope 3 Philosophy
5. Interfaces in Python?
6. Component Architecture
7. ZCML as Glue
8. Page Templates & DTML 2
9. Example: I18n



Let's start with some questions for you!

1. Who uses Zope 2?
2. Who attended last year's talk?
3. Who knew about Zope 3 before?
4. Who is familiar with Python?
5. Who uses other component based systems?

- Web Pages and Applications (CBS New York)
- "Glue Server" binding various services together (Mail logging)
- Server Management (Monitoring Services)
- Object-oriented storage for any application (ZODB)
- Examples: Mailman 3, Games, Web Mail Clients

It really is an

Operating System for the Network

- Security (many authentication mechanisms)
- Transparent Persistence Framework
- Native Internationalization Support
- Through-the-Web Development
- Simple Templating Languages (DTML and ZPT)
- Native Relational Database Integration
- Object-oriented programming for network applications
- Many 3rd party products, that extend Zope in many ways

- Supported RDBs:
 - Oracle, Sybase, DB2, Informix, mySQL, PostGreSQL, ODBC, SAP DB
- Authentication Data Sources:
 - ZODB, RDB Table, LDAP, IMAP, SMB, /etc/passwd
- Some Supported Network Protocols:
 - HTTP(S), FTP, WebDAV, SMTP, IMAP, POP, XML-RPC, SOAP
- Other supported technologies:
 - XML, DOM, COM (Windows only)



Zope Tid Bits (Dismantling myths)

- Zope is **not** a Content Management System
- Zope's competitors include Vignette, BEA Weblogic or IBM Webshpere
- Zope does **not** try to compete with Perl CGI or PHP
- Zope supports all the "batteries" that are available in Python
- Zope is more secure than most people think!

- Provide a complete Web platform
- Utilize a simple scripting language (Python)
- Provide object-oriented storage capabilities (ZODB)
- Provide out-of-the-box security
- Separate tasks between Designer, Web and Application Developer
- Allow full control over the Web (ZMI)
- Make it easy to extend

- Allow non-Zope 3 specific Python code to be used
- Separate User Roles (Designer, Site Admin, Programmer)
- Define functionality through Interfaces
- One object/class is responsible for one thing only
- Explicit is better than implicit (namespaces)
- Develop well-designed code
- Incorporate the lessons-learned from Zope 2

- A simple Python Package independent of Zope
- An Interface is **not** a class!
- Allows us to make a "contract" with a class that implements the interface
- Communication between objects is based on Interfaces
- Example:

```
from Interface import Interface
from Interface.Attribute import Attribute

class ISample(Interface):
    """A sample interface."""

    attr = Attribute("""This is an attribute.""")

    def provideMethod():
        """This is the documentation for the method."""
```

- Categorize objects by functionality
- Allow small parts of the entire Zope 3 framework to be switched with customized or self-written components (for example use Twisted for Server support)
- Distribute sub-packages (such as the Server or I18n package)
- Make it easier to communicate with other component architecture systems

- This object contains only data
- Its methods deal only with manipulating and returning the object's data
- Need to subclass Persistent for ZODB storage
- Examples: Image, File, Folder, Contact ...
- Code Example:

```
from IContentObject import IContentObject
from Persistence import Persistent

class Content(Persistent):

    __implements__ = IContentObject

    attr = None

    def setattr(self, attr):
        """See IContentObject for Documentation"""
        self.attr = attr
```

- This object only provides functionality, no new data
- An adapter implements one interface and provides another
- Example: GZIP a File
- Code Example:

```
from IContentObject import IContentObject
from IConvertAttr import IConvertAttr
```

```
class ConvertAttr:
    __implements__ = IConvertAttr
    __used_for__ = IContentObject
```

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

```
def convertAttr(self):
    attr = self.context.attr
    # do some conversion ....
    return attr
```

- This object only provides functionality, no new data
- An adapter implements one interface and provides another
- Example: View edit screen of a content object
- Code Example:

```
from IContentObject import IContentObject
from Zope.Publisher.Browser.IBrowserView import IBrowserView

class ContentView:
    __implements__ = IBrowserView
    __used_for__ = IContentObject

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

    def viewAttr(self):
        self.request.response.setHeader('Content-type',
                                         'text/plain')
        return self.context.attr
```

- Services provide functionality independent of other objects (even though they sometimes require input data)
- Services can be replaced; the registry knows how to pick the right one.
- Examples: I18n, Roles, Events
- Example on how to use the I18n Service:

```
from ComponentArchitecture import getService

class Foo:

    def method(self):
        service = getService(self, 'Translation Service')
        domain = service.getDomain('Zope 3')
        result = domain.translate('This is a message',
                                 target_lang='de')

        print result
```

Output:

```
Dies ist eine Nachricht
```

- Utilities are like Services, but they are not mission critical
- Utilities will be important for TTW development
- Examples: ??? (I think there are none so far)

- ZCML stands for Zope Configuration Markup Language
- Registers classes in the right component registries
- Tells Zope how the components interact
- Meta directives define the available directives for the user
- Example:

```
<zopeConfigure xmlns="http://namespaces.zope.org/zope"
               xmlns:browser="http://namespaces.zope.org/browser">
  <permission id="Zope.AddFiles" title="Add Files" />

  <content class=".ContentObject.">
    <factory id="ContentObject" permission="Zope.ManageContent"
            title="Content Object" description="A Content Object" />
    <require permission="Zope.ManageContent"
              interface=".IContentObejct." />
  </content>

  <browser:view for=".IContentObject." permission="Zope.View"
               factory="ContentView.">
  </browser:view>
</zopeConfigure>
```

- There will be two Templating Languages:
 - Page Templates
 - Document Templating Markup Language 2 (DTML2)
- They can be either File or Web based
- Both languages are precompiled
- Both languages are Zope independent
- However, the Zope 3 standard are the Page Templates

- Page Templates are an attribute-based language
- Designed to be editable in Designer Tools (Dreamweaver)
- Macros are supported to replace parts of the Template
- Full I18n support (wait for the example later on)
- Example:

```
<html metal:use-macro="views/standard_macros/page">
  <body i18n:domain="example">
    <table>
      <tal:block repeat="user context/users">
        <tr class="odd" tal:condition="repeat/user/odd">
          <td i18n:translate="">The user's name is
            <span tal:replace="user" i18n:name="name">Stephan</span>.
          </td></tr>
        <tr class="even" tal:condition="repeat/user/even">
          <td i18n:translate="">The user's name is
            <span tal:replace="user" i18n:name="name">Stephan</span>.
          </td></tr>
        </tal:block>
      </table>
    </body>
  </html>
```

- DTML is tag-based language (not fully XML compliant)
- Exists as long as Zope does
- In DTML 2 all expressions will be replaced with PT expressions
- DTML 2 will have only explicit namespaces
- Example (science-fiction):

```
<dtml-var "views/page_header" />
<table>
  <dtml-in "user context/users">
    <dtml-if "in/user/odd">
      <tr class="odd">
    <dtml-else>
      <tr class="even">
    </dtml-if>
      <td><dtml-translate>The user's name is
        <dtml-var "user" i18n_name="name">.</dtml-translate>
      </td>
    </tr>
  </tal:block>
</table>
<dtml-var "views/page_footer" />
```

- There will be something like ZClasses NG
- All TTW components will be located in Packages
- One will be able to create new Content objects, Views, Utilities and probably all other component types
- TTW Components will be inserted into the framework using Configuration objects
- We also hope that pure Python code can live inside the ZODB (PythonLabs works on persistent Modules at the moment)

What are the required components in the Zope 3 framework?

- Translation Service (responsible for providing a translation interface)
- Message Catalogs (a storage mechanism for translations - optional)
- Language Negotiator (Service that determines the language to use)
- Charset Negotiator (Service that determines the output character set)

- I18nAware objects (objects that have content in more than one language)
- I18n Resource Directives (let's you create multilingual resources)
- I18n ZPT Namespace
- I18n DTML Tags

And now to the demo... The JobBoard!

What is the JobBoard? The JobBoard example was developed during the PythonLabs sprint. It is intended to show demonstrate a minimal product using the Zope 3 framework.

After the initial version of the I18n support was completed (2.5 weeks ago), Barry Warsaw internationalized and localized the example.

And here it comes...



Questions?

Questions, Comments, Remarks?