Bebop Protocols

Uwe Oestermeier

EuroPython 2007

Overview

Use cases

Shortcomings of Zope3

Configuration

Extensions

Generic functions

zope.fssync

Use Cases at the KMRC

Bebop as a core application

Many experimental prototypes and variants

Rapid Prototyping with Zope3?

Integration with existing code

Evolution vs. revolution

Why is Zope3 difficult to learn?

It's inherently complex: many concepts, many adapters, many ZCML statements, many packages, many dependencies, a long history, numerous add-ons,...

Strict separation of interfaces, code, and configuration

It ▪ 's difficult to work in a copy, paste, modify style

Trends in Zope3

Application to library transition

The big tree vs. small satellites

Many independent versioning schemes

Splitted release manager responsibilities

Will these trends make Zope3 easier?

Will Zope3 be understandable without tacit knowledge?

Follow this man?

Grok smashes ZCML, Bebop generates ZCML

Grok says: Don‘t repeat yourself
Jim says: Explict is better than implict
Bop says: Let the computer do the repetition

Main idea: ZCML directly translated into Python

Stepwise integration into existing applications

No extra learning curve

Hello Bop

```
from persistent import Persistent
from bebop import bop

class World(Persistent):
    pass

greet = bop.GenericFunction('IGreet')
@greet.when(World)
def greet_world(world):
    return 'world'

class Greeting(bop.BrowserView):
    bop.browser.page(World, name="greet", permission='zope.Public')

    def __call__(self):
        return "Hello %s" % greet(self.context)
```

Shortcuts

Generic functions

Decorators

Class advisors

Activate Bop...

```
<configure xmlns="http://iwm-kmrc.de/bebop">
  <protocol
      module="bebop.protocol.demo.hello"
      record="protocol.zcml"/>
</configure>
```

1. Scan a module or package for all declarations
2. Perform the configuration actions (with conflict detection)
3. Record the ZCML for documentary purposes (optional)

The ZCML can be used as a template for overrides.zcml

Bop generates ZCML

```
<configure
      xmlns:browser="http://namespaces.zope.org/browser"
      xmlns:zope="http://namespaces.zope.org/zope"
     >
  <!-- GENERATED PROTOCOL. DO NOT MODIFY OR INCLUDE -->
  <browser:page
    class="bebop.protocol.demo.hello.allinone.Greeting"
    layer="zope.publisher.interfaces.browser.IDefaultBrowserLayer"
    for="bebop.protocol.demo.hello.allinone.World"
    permission="zope.Public"
    name="greet"
    attribute="__call__"
  />
  <zope:adapter
    factory="bebop.protocol.demo.hello.allinone.greet_world"
    provides="bebop.protocol.demo.hello.allinone.IGreet"
    for="bebop.protocol.demo.hello.allinone.World"
  />
</configure>
```

Bop is as short as Grok

```
from persistent import Persistent
from bebop import bop

class World(Persistent):
    bop.application()
    __name__ = 'world'

@bop.view.when(World, name='greet')
def hello_world(world, request):
    return 'Hello %s' % bop.name(world)
```

Works with standard base classes

Registers World as an application factory

Predefined generic functions

No conventions, no base classes ZCML on demand

Bebop Protocols

Inspired by Tim Hochberg‘s proposal on the Python-3000 list

Combines the idea of Python configuration with extensible callables

How does it work?

Protocols

- collect declarations within modules, i.e. specifications how components should be registered
- are able to register and unregister components
- are able to record the declarations in ZCML
- are able to perform the ZCML configuration actions
- are activated with a single ZCML statement

- provide an overwritable __call__ method

Generic Functions

Easy to define in Zope3: register functions as adapters

```
class GenericFunction(AdapterProtocol):
    def __call__(self, *args):
        if len(args) > 1:
            return getMultiAdapter(args, self.provides)
        return self.provides(args[0])
    def when(self, *types, **kw):
        def decorator(f):
            self.declare(factory=f,
                         for_=types,
                         provides=self.provides,
                         permission=kw.get('permission'))
            return f
        return decorator
```

The Interface

```
class IProtocol(zope.interface.Interface):
    def __call__(*args, **kw):
        ...
    def declare(self, *args, **kw):
        ...
    def configure(context, modules=None):
        ...
    def activate(modules=None):
        ...
    def deactivate(modules=None):
        ...
    def record(self, context=None, modules=None):
```

Variants of Generic Functions

- Generic Functions that generate their own interfaces
- Functions that provide a default behavior
- Specializations for view(context, request) functions
- Functions that provide before and after hooks
- and much more

```
title = GenericFunction('bebop.bop.ITitle')
@title.when(zope.interface.Interface)
def default_title(obj):
    dc = IZopeDublinCore(obj, None)
    if dc is None:
        return u'Untitled'
    return dc.title
```

Use of Generic Functions

- Can replace adapters only one or two methods
- Rule of thumb: use them if noun (adapter class name) and verb (method name) express the same basic idea

ITraverser.traverse
INameChooser.chooseName
IRenderer.render

Beyond “hello world”: zope.fssync

Simple idea: export objects to the filesystem, edit them offline, reimport them

Different

- use cases (content management, complete data export/import)
- serialization formats (xml pickles, application formats)
- repositories (directories, archives, svn checkouts)
- security policies, etc.

Non trivial problems: case-insensitive filesystems, unicode filenames, class-based adapter lookup, platform differences

Serialization and deserialization

- How can you be sure that your data are serialized completely?
- Standard interface/type based adapters are risky

```
class IPerson(Interface):
    name = Attribute()
class IEmployee(IPerson):
    insurance_id = Attribute()

class PersonSerializer(Serializer):
    adapts(IPerson)
    def serialize(self):
            ... serializes only the name
```

- Using class based adapters and a default serializer (e.g. xml pickle) would solve the problem

Current Solution: Named utilities

- Register adapter factories as named utilities with the dotted class name as lookup key
- Provide a default adapter that is used if no class-based adapter can be found

```
def synchronizer(obj):
    dn = dottedname(obj.__class__)
    factory = queryUtility(ISynchronizerFactory, name=dn)
    if factory is None:
        factory = queryUtility(ISynchronizerFactory)
    if factory is None:
            raise MissingSynchronizer(dn)
    return factory(obj)
```

Example for zope.app.file

```
class FileSynchronizer(synchronizer.Synchronizer):
    interface.implements(interfaces.IFileSynchronizer)

    def metadata(self):
        md = super(FileSynchronizer, self).metadata()
        if not self.context.contentType or \
            not self.context.contentType.startswith('text'):
            md['binary'] = 'true'
        return md

    def extras(self):
        return synchronizer.Extras(contentType=self.context.contentType)

    def dump(self, writeable):
        ...

    def load(self, readable):
        ...
```

Shortcomings of zope.fssync

Differences and commonalities are difficult to handle

- SVN stores metadata and annotations different from a Zip archive
- Binary file content is the same in all target system
- Some aspects (e.g. filenames) are platform dependent (case-insensitivity, platform specific unicode encodings)
- Others are application specific (e.g. the basic serialization format)

Proposed Solution

A bunch of generic functions

- some simple adapters, some multi-adapters
- some class-based, some (super-)type based

Different lookup strategies by different __call__ methods

The basic syntax is always the same:

```
@metadata.when(zope.app.interfaces.IFile, ISVNRepository)
def metadata_file_svn(file, repository):
    ...

@serialize.when(zope.app.File, IWriteable)
def serialize_file(file, writeable):
  ...
```

PEP 3124 by Phillip J. Eby

Proposes a standard module ``overloading`` with generic functions, interfaces, and adaptation

Consensus of the discussion

- generic functions should be part of Python
- other extensions (like @before and @after decorators) will probably not be part of the standard libraries

Zope3 should have a variant of generic functions that anticipates Pythons future

Specializations of generic functions can handle other aspects of Eby's proposal, e.g. @before and @after decorators

Current state of bebop.protocol

Actually used in our work on Bebop

Protocols for adapters, utilities, subscribers, browser pages, generic functions

Some directives are still missing (zope:view, browser:menu, ...)

Some aspects (e.g. generic functions) should be part of the Zope3 component architecture

Sources

Cheeseshop:

```
Error...

There's been a problem with your request

exceptions.AttributeError: SMTP instance has no
  attribute 'sock'
```

In the meanwhile:

- http://svn.kmrc.de/projects/devel/bebop.protocol/
- http://svn.zope.org/zope.fssync/trunk/src/zope/fssync/generic.txt