

Web Application Development Using Python and Zope Components

Baiju Muthukadan
Paul Carduner

Version: 0.1.2.8
Project Home: <https://launchpad.net/wadupaz>
Printed Book: <http://www.lulu.com/content/2000326>
Online PDF: <http://wadupaz.muthukadan.net/wadupaz.pdf>

Copyright 2008 Baiju Muthukadan <baiju.m.mail AT gmail.com>

Copyright 2008 Paul Carduner <paulcarduner AT gmail.com>

This work is licensed under the Creative Commons Attribution-ShareAlike License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/3.0/> or send a letter to Creative Commons, 559 Nathan Abbott Way, Stanford, California 94305, USA. A summary of the license is given below.

You are free:

- to Share - to copy, distribute and transmit the work
- to Remix - to adapt the work

Under the following conditions:

- Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.
- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to this web page.
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.

Preface

Our aim is to produce a book about web application development using Python and Zope components. Please send your suggestions/comments to: *baiju.m.mail AT gmail.com* or *paulcarduner AT gmail.com* .

We use the term “Zope components” loosely; this book will actually explain how to use Zope 3 packages to develop web applications. This book treats Zope 3 as a set of frameworks and small libraries.

This book has borrowed content from Stephan Richter’s training material. Thank you very much Stephan for providing it.

Baiju Muthukadan & Paul Carduner

Contents

Preface	iii
1 Getting Started	1
1.1 Brief history	2
1.2 Installation	2
1.2.1 Installing Python	2
1.2.2 Buildout configuration	3
1.2.3 Setting up a development sandbox	4
1.3 A simple application	5
1.3.1 Configuring application	5
1.3.2 Running application	7
1.3.3 Using ZMI	8
1.4 Hello world	8
1.5 Summary	9
1.6 Discussion	9
2 Development Tools	11
2.1 Eggs	11
2.2 Buildout	11

2.2.1	Recipes	12
2.2.2	Using a recipe	13
2.2.3	Developing a package	13
2.3	Summary	15
3	Component Architecture	17
3.1	Installation	18
3.2	Interfaces	18
3.2.1	Declaring interfaces	20
3.2.2	Implementing interfaces	21
3.3	Adapters	22
3.3.1	Implementation	22
3.3.2	Registration	23
3.3.3	Querying adapter	24
3.3.4	Retrieving adapter using interface	26
3.4	Utility	27
3.4.1	Simple utility	27
3.4.2	Named utility	28
4	Testing	31
4.1	Unit testing	31
4.1.1	Running tests	33
5	Browser Resources	35
5.1	File Resource	35
5.2	Image Resource	36
5.3	Directory Resource	37
5.4	ZRT Resource	37

CONTENTS

vii

6 Pages 39

7 Content Components 41

8 Internationalization and Localization 43

Chapter 1

Getting Started

Zope 3 is a set of frameworks and small libraries for web application development. This chapter will go through the history of Zope. Then, you will learn about the installation of Zope 3 in detail.

The term *ZOPE* is an acronym for *Z Object Publishing Environment* (the *Z* doesn't really mean anything in particular). However, nowadays *ZOPE* is simply written as *Zope*.

The Zope community has produced many frameworks and libraries for developing web applications. The first among these framework was **Bobo** – an object publishing system. Later, along with an object database (**ZODB**), a templating language (**DTML**) and some other technologies, Zope Corporation and the Zope community released Zope. Over the years, the Zope community produced many new projects ranging from content management systems (**CMF**), to a pluggable authentication system (**PAS**). Of particular importance was the creation of the Zope component architecture (**ZCA**), which lead to a complete rewrite of Zope known as Zope 3. Most recently, in 2007 the Zope community created yet another framework based on Zope 3 called **Grok**. The original Zope which is now known as Zope 2 is also widely used.

1.1 Brief history

The beginning of Zope's story goes something like this, in 1996, Jim Fulton was drafted to teach a class on common gateway interface (CGI) programming, despite not knowing very much about the subject. CGI programming is a commonly-used web development model that allows developers to construct dynamic websites. On his way to the class, Jim studied all the existing documentation on CGI. On the way back, Jim considered what he didn't like about traditional, CGI-based programming environments. From these initial musings, the core of Zope was written while flying back from the CGI class.

Zope Corporation (then known as Digital Creations) went on to release three open-source software packages to support web publishing: Bobo, Document Template, and BoboPOS. These packages were written in a language called Python, and provided a web publishing facility, text templating, and an object database, respectively. Digital Creations developed a commercial application server based on their three open-source components. This product was called Principia. In November of 1998, investor Hadar Pedhazur convinced Digital Creations to open source Principia. These packages evolved into what are now the core components of Zope 2.

In 2001, the Zope community began working on a component architecture for Zope, but after several years they ended up with something much more: Zope 3. While Zope 2 was powerful and popular, Zope 3 was designed to bring web application development to the next level.

1.2 Installation

1.2.1 Installing Python

The Zope community has always recommended using a custom built Python for development and deployment. Python 2.4 is the recommended version for Zope 3, although Python 2.5 will also work but is not yet officially supported. To install Python, you will be required to install *gcc*, *g++* and other development tools on your system. A typical installation of Python can be done like this:

```
$ wget -c http://python.org/ftp/python/2.4.4/Python-2.4.4.tar.bz2
```

```
$ tar jxvf Python-2.4.4.tar.bz2
$ cd Python-2.4.4
$ ./configure --prefix=/home/baiju/usr
$ make
$ make install
```

While configuring, as given above, you can give a *prefix* to install Python in that location. The above steps will install Python inside */home/baiju/usr* directory. You can run the Python interpreter like this:

```
$ ~/usr/bin/python2.4
>>> print "Hello, world!"
Hello, world!
```

Note: libreadline and zlib

If you are not getting old statements in Python interactive prompt using up-arrow key, try installing *libreadline* development libraries (Hint: *apt-cache search readline*). After installing this library, you have to install Python again.

You will also be required to install *zlib* (Hint: *apt-cache search zlib compression library*) to properly install Zope 3.

1.2.2 Buildout configuration

Traditionally, Zope 3 was released as a tar ball. However, starting with version 3.4, Zope 3 was split into many packages. This book will use a build tool called Buildout for developing Zope 3 applications.

The default configuration for Buildout will be stored in the *.buildout* directory of your home directory with the file name *default.cfg*. You can add the following to your *\$HOME/.buildout/default.cfg* file:

Listing 1.1: *\$HOME/.buildout/default.cfg*

```
1 [buildout]
2 newest = false
3 eggs-directory = /home/baiju/eggs
4 find-links = http://download.zope.org/ppix
```

The *eggs-directory* is where Buildout stores the eggs that are downloaded. The last option, *find-links* points to a reliable mirror of the Python Package Index (PyPI).

1.2.3 Setting up a development sandbox

To demonstrate the concepts, tools and techniques, we are going to develop a simple ticket/issue tracking application called “Ticket Collector”. To begin the work, first create a directory for the project. After creating the directory, create a *buildout.cfg* file as given below. To bootstrap this application checkout *bootstrap.py* and run it inside that directory.

```
$ mkdir ticketcollector
$ cd ticketcollector
$ echo "#Buildout configuration" > buildout.cfg
$ svn co svn://svn.zope.org/repos/main/zc.buildout/trunk/bootstrap
$ ~/usr/bin/python2.4 bootstrap/bootstrap.py
```

You can see a *buildout* script created inside *bin* directory. Now onwards, you can run this script when changing Buildout configuration.

Note

You can save *bootstrap.py* in a local repository. If you are using svn for managing repository, create an svn:external to the svn URL given above.

Our application is basically a Python package. First we will create an 'src' directory to place our package. Inside the 'src' directory, you can create 'ticketcollector' Python package. You can create the 'src' and the 'ticketcollector' package like this:

```
$ mkdir src
$ mkdir src/ticketcollector
$ echo "#Python package" > src/ticketcollector/__init__.py
```

To start building our package you have to create a *setup.py* file. The *setup.py* should have the minimum details as given below:

Listing 1.2: setup.py

```
1 from setuptools import setup, find_packages
3 setup(
```

```
4     name='ticketcollector',
5     version='0.1',

7     packages=find_packages('src'),
8     package_dir={'': 'src'},

10    install_requires=['setuptools',
11                      'zope.app.zcmlfiles',
12                      'zope.app.twisted',
13                      'zope.app.securitypolicy',
14                      ],
15    include_package_data=True,
16    zip_safe=False,
17    )
```

We have included bare minimum packages required for installation here: *zope.app.zcmlfiles*, *zope.app.twisted* and *zope.app.securitypolicy*.

Modify *buildout.cfg* as given below:

Listing 1.3: buildout.cfg

```
1 [buildout]
2 develop = .
3 parts = py

5 [py]
6 recipe = zc.recipe.egg
7 eggs = ticketcollector
8 interpreter = python
```

Now run *buildout* script inside *bin* directory. This will download all necessary eggs and install it. So installing Zope is nothing but just setting up a *buildout* with *setup.py* with required packages (*install_requires*) for installation. Unless you specified a *parts* section which use *ticketcollector* in some way, buildout will not download dependency packages.

1.3 A simple application

1.3.1 Configuring application

To run the Zope, you have to use a buildout recipe, which will be explained in next chapter. Here we are going to use *zc.zope3recipes:app* recipe for setting up our application, *ticketcollector*.

Listing 1.4: buildout.cfg

```

1 [buildout]
2 develop = .
3 parts = ticketcollectorapp instance

5 [zope3]
6 location =

8 [ticketcollectorapp]
9 recipe = zc.zope3recipes:app
10 site.zcml =
11 <include package="ticketcollector" file="application.zcml" />
12 eggs = ticketcollector

14 [instance]
15 recipe = zc.zope3recipes:instance
16 application = ticketcollectorapp
17 zope.conf = ${database:zconfig}

19 [database]
20 recipe = zc.recipe.filestorage

```

Then create *application.zcml* inside *src/ticketcollector* directory with the following text. Consider it as boiler plate code now, which we will go through later in chapter 3.

Listing 1.5: src/ticketcollector/application.zcml

```

1 <configure
2   xmlns="http://namespaces.zope.org/zope"
3   i18n_domain="zope"
4   >
5 <include package="zope.app.securitypolicy" file="meta.zcml" />

6
7 <include package="zope.app.zcmlfiles" />
8 <include package="zope.app.authentication" />
9 <include package="zope.app.securitypolicy" />
10 <include package="zope.app.twisted" />

11
12 <securityPolicy
13   component="zope.app.securitypolicy.zopepolicy.ZopeSecurityPolicy" />

14
15 <role id="zope.Anonymous" title="Everybody"
16   description="All users have this role implicitly" />
17 <role id="zope.Manager" title="Site Manager" />
18 <role id="zope.Member" title="Site Member" />

19
20 <grant permission="zope.View"
21   role="zope.Anonymous" />
22 <grant permission="zope.app.dublincore.view"

```

```
23   role="zope.Anonymous" />
25 <grantAll role="zope.Manager" />
27 <unauthenticatedPrincipal
28   id="zope.anybody"
29   title="Unauthenticated User" />
31 <unauthenticatedGroup
32   id="zope.Anybody"
33   title="Unauthenticated Users" />
35 <authenticatedGroup
36   id="zope.Authenticated"
37   title="Authenticated Users" />
39 <everybodyGroup
40   id="zope.Everybody"
41   title="All Users" />
43 <principal
44   id="zope.manager"
45   title="Manager"
46   login="admin"
47   password_manager="Plain Text"
48   password="admin"
49 />
51 <grant
52   role="zope.Manager"
53   principal="zope.manager" />
55 </configure>
```

1.3.2 Running application

Now you can run the application by running *buildout* script followed by *instance* script.

```
$ ./bin/buildout
$ ./bin/instance fg
```

So running Zope is nothing but just using a *buildout* recipe with proper configuration.

1.3.3 Using ZMI

After running your instance, if you open a web browser and go to `http://localhost:8080` you'll see the ZMI (*Zope Management Interface*).

Go ahead and click the Login link at the upper right. Enter the user name and password you gave when creating the instance. Now click on *[top]* under Navigation on the right. Play around with adding some content objects (the Zope 3 name for instances that are visible in the ZMI). Note how content objects can be arranged in a hierarchy by adding **folders** which are special content objects that can hold other content objects.

There is nothing special about the ZMI, it is just the default skin for Zope 3. You can modify it to your liking, or replace it entirely.

When you're done exploring with the ZMI, go back to the window where you typed `runzope` and see that each request from your browser was displayed there as it happened. Press Control-C to stop Zope.

1.4 Hello world

Now you can begin your development inside `src/ticketcollector` directory. Create a `browser.py` with following content:

Listing 1.6: `src/ticketcollector/browser.py`

```
1 from zope.publisher.browser import BrowserView
2
3 class HelloView(BrowserView):
4
5     def __call__(self):
6         return """
7         <html>
8         <head>
9             <title>Hello World</title>
10        </head>
11        <body>
12            Hello World
13        </body>
14        </html>
15        """
```

Now append the following text just above the last line of `application.zcml`:

Listing 1.7: src/ticketcollector/application.zcml

```
56 <browser:page
57   for="*"
58   name="hello"
59   permission="zope.Public"
60   class="ticketcollector.browser.HelloView"
61 />
```

As you can see above, we are using *page* attribute from the *browser* namespace. So, you have to include the namespace in the beginning ZCML as shown below:

Listing 1.8: src/ticketcollector/application.zcml

```
1 <configure
2   xmlns="http://namespaces.zope.org/zope"
3   xmlns:browser="http://namespaces.zope.org/browser"
4   i18n_domain="zope"
5   >
```

Now access *http://localhost:8080/hello*, you can see that it displaying “Hello World”.

1.5 Summary

This chapter started with a brief history of Zope, then we moved to setting up a development sandbox. Later we ended with a hello world application.

1.6 Discussion

1. Have you thought about using *zc.buildout* inside *virtualenv* ?
2. Do you think installing *setuptools* and *zc.buildout* in Python’s sitepackages is a bad idea?
3. How do you feel about isolated working environment while developing application?
4. Registration of components can be done in Python code or ZCML, which one you will prefer? What about keeping a balance between what should be done in Python and ZCML ? (If you don’t understand this question, think about it after few chapters)

Chapter 2

Development Tools

Before going to the details about how to develop a web application using Python and Zope components, we should familiarize some tools. In this chapter you will about Python eggs and buildouts.

2.1 Eggs

Eggs are Python's new distribution format managed using *setuptools* package ¹.

To install an egg, you can use `easy_install` program ².

2.2 Buildout

From the Buildout documentation:

The Buildout project provides support for creating applications, especially Python applications. It provides tools for assembling applications from multiple parts, Python or otherwise. An application may actually contain multiple programs, processes, and configuration settings.

¹<http://peak.telecommunity.com/DevCenter/PythonEggs>

²<http://peak.telecommunity.com/DevCenter/EasyInstall>

The word “buildout” refers to a description of a set of parts and the software to create and assemble them. It is often used informally to refer to an installed system based on a buildout definition. For example, if we are creating an application named “Foo”, then “the Foo buildout” is the collection of configuration and application-specific software that allows an instance of the application to be created. We may refer to such an instance of the application informally as “a Foo buildout”.

Buildout provides support for creating, assembling and deploying applications, especially Python applications. You can build applications using Buildout recipes. Recipes are Python programs which follows a pattern to build various parts of an application. For example, a recipe will install Python eggs and another one will install test runner etc. Applications can be assembled from multiple parts with different configurations. A part can be a Python egg or any other program. We have already seen how to use buildout to setup a Zope 3 application in the getting started chapter.

2.2.1 Recipes

Buildout recipes are distributed in egg formats. Some examples of recipes are:

- **zc.recipe.egg** – The egg recipe installes one or more eggs, with their dependencies. It installs their console-script entry points with the needed eggs included in their paths.
- **zc.recipe.testrunner** – The testrunner recipe creates a test runner script for one or more eggs.
- **zc.recipe.zope3recipes** – Recipes for creating Zope 3 instances with distinguishing features:
 - Don’t use a skeleton
 - Separates application and instance definition
 - Don’t support package-includes
- **zc.recipe.filestorage** – The filestorage recipe sets up a ZODB file storage for use in a Zope 3 instance creayed by the *zope3recipes* recipe.

2.2.2 Using a recipe

The procedure for using a recipe is common for almost all recipes. You can create buildouts with parts controlled recipes. Suppose you want to experiment with one package, say, *zope.component*, you can use *zc.recipe.egg* for installing it in buildout. The *zc.recipe.egg* will also provide an interpreter with the egg installed in the path. First you can create a directory and initialize Buildout:

```
$ mkdir explore-zope.component
$ cd explore-zope.component
$ echo "#Buildout configuration" > buildout.cfg
$ svn co svn://svn.zope.org/repos/main/zc.buildout/trunk/bootstrap
$ ~/usr/bin/python2.4 bootstrap/bootstrap.py
```

Now modify the *buildout.cfg* like this:

```
[buildout]
parts = py

[py]
recipe = zc.recipe.egg
interpreter = mypython
eggs = zope.component
```

Now run *buildout* script inside *bin* directory. This will download *zope.component* and its dependency eggs and install it. Now you can access the interpreter created by the Buildout recipe like this:

```
$ ./bin/buildout
$ ./bin/mypython
>>> import zope.component
```

2.2.3 Developing a package

The initial steps are not different from the above example:

```
$ mkdir hello
$ cd hello
$ echo "#Buildout configuration" > buildout.cfg
$ svn co svn://svn.zope.org/repos/main/zc.buildout/trunk/bootstrap
$ ~/usr/bin/python2.4 bootstrap/bootstrap.py
```

Our application is a simple hello world package. First we will create an 'src' directory to place our package. Inside the 'src' directory, you can create the 'hello' Python package. You can create the 'src' and the 'hello' package like this:

```
$ mkdir src
$ mkdir src/hello
$ echo "#Python package" > src/hello/__init__.py
```

Now create a file named *say.py* inside the *hello* package with this code:

```
def say_hello():
    print "Hello"
```

To start building our package you have to create a *setup.py* file. The *setup.py* should have the minimum details as given below:

```
from setuptools import setup, find_packages

setup(
    name='hello',
    version='0.1',

    packages=find_packages('src'),
    package_dir={'': 'src'},

    install_requires=['setuptools',
                     ],
    entry_points = {'console_scripts':
                    ['print_hello = hello.say:say_hello']},
    include_package_data=True,
    zip_safe=False,
)
```

Modify *buildout.cfg* as given below:

```
[buildout]
develop = .
parts = py

[py]
recipe = zc.recipe.egg
scripts = print_hello
eggs = hello
```

Now run *buildout* script inside *bin* directory. Now you can run the *print_hello* script.

```
$ ./bin/buildout
$ ./bin/print_hello
Hello
```

2.3 Summary

This chapter provided a brief introduction to eggs. Later we found how to use buildout tool developing application.

Chapter 3

Component Architecture

Zope Component Architecture (ZCA) is a framework for supporting component based design and programming. It is very well suited to developing large Python software systems. The ZCA is not specific to the Zope web application server: it can be used for developing any Python application.

The ZCA is all about using Python objects effectively. Components are reusable objects with introspectable interfaces. A component provides an interface implemented in a class, or any other callable object. It doesn't matter how the component is implemented, the important part is that it comply with its interface contracts. Using ZCA, you can spread the complexity of systems over multiple cooperating components. It helps you to create two basic kinds of components: *adapter* and *utility*.

There are two core packages related to the ZCA:

- **zope.interface** is used to define the interface of a component.
- **zope.component** deals with registration and retrieval of components.

Remember, the ZCA is not about the components themselves, rather it is about creating, registering, and retrieving components. Remember also, an *adapter* is a normal Python class (or a factory in general) and *utility* is a normal Python callable object.

The ZCA framework is developed as part of the Zope 3 project. As noted earlier, it is a pure Python framework, so it can be used in any kind of Python application. Currently

both Zope 3 and Zope 2 projects use this framework extensively. There are many other projects including non-web applications using it.

3.1 Installation

Using **zc.buildout** with **zc.recipe.egg** recipe you can create Python interpreter with specified Python eggs. First you can create a directory and initialize Buildout:

```
$ mkdir explore-zope.component
$ cd explore-zope.component
$ echo "#Buildout configuration" > buildout.cfg
$ svn co svn://svn.zope.org/repos/main/zc.buildout/trunk/bootstrap
$ ~/usr/bin/python2.4 bootstrap/bootstrap.py
```

Now modify the *buildout.cfg* like this:

```
[buildout]
parts = py

[py]
recipe = zc.recipe.egg
eggs = zope.component
interpreter = mpython
```

Now run *buildout* script inside *bin* directory. This will download *zope.component* and its dependency eggs and install it. Now you can access the interpreter created by the Buildout recipe like this:

```
$ ./bin/buildout
$ ./bin/mpython
>>> import zope.component
```

3.2 Interfaces

The README.txt of *zope.interface* package defines interfaces like this:

Interfaces are objects that specify (document) the external behavior of objects that "provide" them. An interface specifies behavior through:

- Informal documentation in a doc string
- Attribute definitions
- Invariants, which are conditions that must hold for objects that provide the interface

The classic software engineering book *Design Patterns*¹ by the *Gang of Four* recommends that you "Program to an interface, not an implementation". Defining a formal interface is helpful in understanding a system. Moreover, interfaces bring to you all the benefits of ZCA.

An interface specifies the characteristics of an object, it's behaviour, it's capabilities. The interface describes *what* an object can do, to learn *how*, you must look at the implementation.

Commonly used metaphors for interfaces are *contract* or *blueprint*, the legal and architectural terms for a set of specifications.

In some modern programming languages: Java, C#, VB.NET etc, interfaces are an explicit aspect of the language. Since Python lacks interfaces, ZCA implements them as a meta-class to inherit from.

Here is a classic *hello world* style example::

```
>>> class Host(object):
...     def goodmorning(self, name):
...         """Say good morning to guests"""
...         return "Good morning, %s!" % name
```

¹<http://en.wikipedia.org/wiki/Design.Patterns>

In the above class, you defined a ‘goodmorning’ method. If you call the ‘goodmorning’ method from an object created using this class, it will return ‘Good morning, ...j ::

```
>>> host = Host()
>>> host.goodmorning('Jack')
'Good morning, Jack!'
```

Here, *host* is the actual object your code uses. If you want to examine implementation details you need to access the class *Host*, either via the source code or an API ² documentation tool.

Now we will begin to use the ZCA interfaces. For the class given above you can specify the interface like this:

```
>>> from zope.interface import Interface
>>> class IHost(Interface):
...     def goodmorning(guest):
...         """Say good morning to guest"""
```

As you can see, the interface inherits from `zope.interface.Interface`. This use (abuse?) of Python’s class statement is how ZCA defines an interface. The *I* prefix for the interface name is a useful convention.

3.2.1 Declaring interfaces

You have already seen how to declare an interface using “zope.interface” in previous section. This section will explain the concepts in detail.

Consider this example interface:

```
>>> from zope.interface import Interface
>>> from zope.interface import Attribute
>>> class IHost(Interface):
...     """A host object"""
```

²http://en.wikipedia.org/wiki/Application_programming_interface

```

...
...     name = Attribute("""Name of host""")
...
...     def goodmorning(guest):
...         """Say good morning to guest"""

```

The interface, *IHost* has two attributes, *name* and *goodmorning*. Recall that, at least in Python, methods are also attributes of classes. The *name* attribute is defined using *zope.interface.Attribute* class. When you add the attribute *name* to the *IHost* interface, you don't set an initial value. The purpose of defining the attribute *name* here is merely to indicate that any implementation of this interface will feature an attribute named *name*. In this case, you don't even say what type of attribute it has to be!. You can pass a documentation string as a first argument to *Attribute*.

The other attribute, *goodmorning* is a method defined using a function definition. Note that *self* is not required in interfaces, because *self* is an implementation detail of class. For example, a module can implement this interface. If a module implement this interface, there will be a *name* attribute and *goodmorning* function defined. And the *goodmorning* function will accept one argument.

Now you will see how to connect *interface-class-object*. So object is the real living thing, objects are instances of classes. And interface is the actual definition of the object, so classes are just the implementation details. This is why you should program to an interface and not to an implementation.

Now you should familiarize two more terms to understand other concepts. First one is 'provide' and the other one is 'implement'. Object provides interfaces and classes implement interfaces. In other words, objects provide interfaces that their classes implement. In the above example *host* (object) provides *IHost* (interface) and *Host* (class) implement *IHost* (interface). One object can provide more than one interface also one class can implement more than one interface. Objects can also provide interfaces directly, in addition to what their classes implement.

3.2.2 Implementing interfaces

To declare a class implements a particular interface, use the function *zope.interface.implements* in the class statement.

Consider this example, here *Host* implements *IHost*:

```

>>> from zope.interface import implements

>>> class Host(object):
...     implements(IHost)
...     name = u''
...     def goodmorning(self, guest):
...         """Say good morning to guest"""
...         return "Good morning, %s!" % guest

```

3.3 Adapters

3.3.1 Implementation

This section will describe adapters in detail. Zope component architecture, as you noted, helps to effectively use Python objects. Adapter components are one of the basic components used by Zope component architecture for effectively using Python objects. Adapter components are Python objects, but with well defined interface.

To declare a class is an adapter use *adapts* function defined in *zope.component* package. Here is a new *FrontDeskNG* adapter with explicit interface declaration:

```

>>> from zope.interface import implements
>>> from zope.component import adapts

>>> class FrontDeskNG(object):
...     implements(IDesk)
...     adapts(IGuest)
...     def __init__(self, guest):
...         self.guest = guest
...     def register(self):
...         guest = self.guest
...         next_id = get_next_id()
...         bookings_db[next_id] = {

```

```
...         'name': guest.name,  
...         'place': guest.place,  
...         'phone': guest.phone  
...     }
```

What you defined here is an ‘adapter’ for ‘IDesk’, which adapts ‘IGuest’ object. The ‘IDesk’ interface is implemented by ‘FrontDeskNG’ class. So, an instance of this class will provide ‘IDesk’ interface.

```
>>> class Guest(object):  
...  
...     implements(IGuest)  
...  
...     def __init__(self, name, place):  
...         self.name = name  
...         self.place = place  
  
>>> jack = Guest("Jack", "Bangalore")  
>>> jack_frontdesk = FrontDeskNG(jack)  
  
>>> IDesk.providedBy(jack_frontdesk)  
True
```

The *FrontDeskNG* is just one adapter you created, you can also create other adapters which handles guest registration differently.

3.3.2 Registration

To use this adapter component, you have to register this in a component registry also known as site manager. A site manager normally resides in a site. A site and site manager will be more important when developing a Zope 3 application. For now you only required to bother about global site and global site manager (or component registry). A global site manager will be in memory, but a local site manager is persistent.

To register your component, first get the global site manager:

```
>>> from zope.component import getGlobalSiteManager  
>>> gsm = getGlobalSiteManager()  
>>> gsm.registerAdapter(FrontDeskNG,  
...                     (IGuest,), IDesk, 'ng')
```

To get the global site manager, you have to call *getGlobalSiteManager* function available in *zope.component* package. In fact, the global site manager is available as an attribute (*globalSiteManager*) of *zope.component* package. So, you can directly use *zope.component.globalSiteManager* attribute. To register the adapter in component, as you can see above, use *registerAdapter* method of component registry. The first argument should be your adapter class/factory. The second argument is a tuple of *adaptee* objects, i.e, the object which you are adapting. In this example, you are adapting only *IGuest* object. The third argument is the interface implemented by the adapter component. The fourth argument is optional, that is the name of the particular adapter. Since you gave a name for this adapter, this is a *named adapter*. If name is not given, it will default to an empty string ("").

In the above registration, you have given the adaptee interface and interface to be provided by the adapter. Since you have already given these details in adapter implementation, it is not required to specify again. In fact, you could have done the registration like this:

```
>>> gsm.registerAdapter(FrontDeskNG, name='ng')
```

There are some old API to do the registration, which you should avoid. The old API functions starts with *provide*, eg: *provideAdapter*, *provideUtility* etc. While developing a Zope 3 application you can use Zope configuration markup language (ZCML) for registration of components. In Zope 3, local components (persistent components) can be registered from Zope Management Interface (ZMI) or you can do it programmatically also.

You registered *FrontDeskNG* with a name *ng*. Similarly you can register other adapters with different names. If a component is registered without name, it will default to an empty string.

3.3.3 Querying adapter

Retrieving registered components from component registry is achieved through two functions available in *zope.component* package. One of them is *getAdapter* and the other is *queryAdapter*. Both functions accepts same arguments. The *getAdapter* will raise *ComponentLookupError* if component lookup fails on the other hand *queryAdapter* will return 'None'.

You can import the methods like this:

```
>>> from zope.component import getAdapter
>>> from zope.component import queryAdapter
```

In the previous section you have registered a component for guest object (adaptee) which provides 'IDesk' interface with name as *ng*. In the first section of this chapter, you have created a guest object named *jack*.

This is how you can retrieve a component which adapts the interface of *jack* object (*IGuest*) and provides *IDesk* interface also with name as *ng*. Here both *getAdapter* and *queryAdapter* works similarly:

```
>>> getAdapter(jack, IDesk, 'ng') #doctest: +ELLIPSIS
<FrontDeskNG object at ...>
>>> queryAdapter(jack, IDesk, 'ng') #doctest: +ELLIPSIS
<FrontDeskNG object at ...>
```

As you can see, the first argument should be adaptee then, the interface which should be provided by component and last the name of adapter component.

If you try to lookup the component with an name not used for registration but for same adaptee and interface, the lookup will fail. Here is how the two methods works in such a case::

```
>>> getAdapter(jack, IDesk, 'not-exists') #doctest: +ELLIPSIS
Traceback (most recent call last):
...
ComponentLookupError: ...
>>> reg = queryAdapter(jack,
...                     IDesk, 'not-exists') #doctest: +ELLIPSIS
>>> reg is None
True
```

As you can see above, *getAdapter* raised a *ComponentLookupError* exception, but *queryAdapter* returned *None* when lookup failed.

The third argument, the name of registration, is optional. If the third argument is not given it will default to empty string (""). Since there is no component registered with an empty string, *getAdapter* will raise *ComponentLookupError*. Similarly *queryAdapter* will return *None*, see yourself how it works:

```

>>> getAdapter(jack, IDesk) #doctest: +ELLIPSIS
Traceback (most recent call last):
...
ComponentLookupError: ...
>>> reg = queryAdapter(jack, IDesk) #doctest: +ELLIPSIS
>>> reg is None
True

```

In this section you have learned how to register a simple adapter and how to retrieve it from component registry. These kind of adapters is called single adapter, because it adapts only one adaptee. If an adapter adapts more that one adaptee, then it is called multi adapter.

3.3.4 Retrieving adapter using interface

Adapters can be directly retrieved using interfaces, but it will only work for non-named single adapters. The first argument is the adaptee and the second argument is a keyword argument. If adapter lookup fails, second argument will be returned.

```

>>> IDesk(jack, alternate='default-output')
'default-output'

```

Keyword name can be omitted:

```

>>> IDesk(jack, 'default-output')
'default-output'

```

If second argument is not given, it will raise `TypeError`:

```

>>> IDesk(jack) #doctest: +NORMALIZE_WHITESPACE +ELLIPSIS
Traceback (most recent call last):
...
TypeError: ('Could not adapt',
 <Guest object at ...>,
 <InterfaceClass __builtin__.IDesk>)

```

Here `FrontDeskNG` is registered without name:

```

>>> gsm.registerAdapter(FrontDeskNG)

```

Now the adapter lookup should succeed:

```
>>> IDesk(jack, 'default-output') #doctest: +ELLIPSIS
<FrontDeskNG object at ...>
```

For simple cases, you may use interface to get adapter components.

3.4 Utility

Now you know the concept of interface, adapter and component registry. Sometimes it would be useful to register an object which is not adapting anything. Database connection, XML parser, object returning unique Ids etc. are examples of these kinds of objects. These kind of components provided by the ZCA are called *utility* components.

Utilities are just objects that provide an interface and that are looked up by an interface and a name. This approach creates a global registry by which instances can be registered and accessed by different parts of your application, with no need to pass the instances around as parameters.

You need not to register all component instances like this. Only register components which you want to make replaceable.

3.4.1 Simple utility

A utility can be registered with a name or without a name. A utility registered with a name is called named utility, which you will see in the next section. Before implementing the utility, as usual, define its interface. Here is a *greeter* interface::

```
>>> from zope.interface import Interface
>>> from zope.interface import implements

>>> class IGreeter(Interface):
...
...     def greet(name):
...         """Say hello"""
```

Like an adapter a utility may have more than one implementation. Here is a possible implementation of the above interface::

```
>>> class Greeter(object):
...     ...
...     implements(IGreeter)
...     ...
...     def greet(self, name):
...         return "Hello " + name
```

The actual utility will be an instance of this class. To use this utility, you have to register it, later you can query it using the ZCA API. You can register an instance of this class (‘utility’) using *registerUtility*:

```
>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()

>>> greet = Greeter()
>>> gsm.registerUtility(greet, IGreeter)
```

In this example you registered the utility as providing the ‘IGreeter’ interface. You can look the interface up with either ‘queryUtility’ or *getUtility*:

```
>>> from zope.component import queryUtility
>>> from zope.component import getUtility

>>> queryUtility(IGreeter).greet('Jack')
'Hello Jack'

>>> getUtility(IGreeter).greet('Jack')
'Hello Jack'
```

As you can see, adapters are normally classes, but utilities are normally instances of classes. Only once you are creating the instance of a utility class, but adapter instances are dynamically created whenever you query for it.

3.4.2 Named utility

When registering a utility component, like adapter, you can use a name. As mentioned in the previous section, a utility registered with a particular name is called named utility.

This is how you can register the ‘greeter’ utility with a name::

```
>>> greet = Greeter()
>>> gsm.registerUtility(greet, IGreeter, 'new')
```

In this example you registered the utility with a name as providing the 'IGreeter' interface. You can look up the interface with either 'queryUtility' or 'getUtility':

```
>>> from zope.component import queryUtility
>>> from zope.component import getUtility

>>> queryUtility(IGreeter, 'new').greet('Jill')
'Hello Jill'

>>> getUtility(IGreeter, 'new').greet('Jill')
'Hello Jill'
```

As you can see here, while querying you have to use the 'name' as second argument.

Calling 'getUtility' function without a name (second argument) is equivalent to calling with an empty string as the name. Because, the default value for second (keyword) argument is an empty string. Then, component lookup mechanism will try to find the component with name as empty string, and it will fail. When component lookup fails it will raise *ComponentLookupError* exception. Remember, it will not return some random component registered with some other name. The adapter look up functions, 'getAdapter' and 'queryAdapter' also works similarly.

Chapter 4

Testing

4.1 Unit testing

This chapter will discuss about unit testing and integration testing. Doctest-based testing is heavily used in Zope 3. And test driven development (TDD) is preferred in Zope 3.

To explain the idea, consider a use case. A module is required with a function which returns "Good morning, name!". The name will be given as an argument. Before writing the real code write the unit test for this. In fact you will be writing the real code and it's test cases almost in parallel. So create a file named `example1.py` with the function definition:

```
def goodmorning(name):  
    "This returns a good morning message"
```

See, you have not yet written the logic. But this is necessary to run tests successfully with failures!. Ok, now create a file named `example1.txt` with test cases, use reStructuredText format:

These are tests for `example1` module.

First import the module:

```
>>> import example1
```

Now call the function *goodmorning* without any arguments:

```
>>> example1.goodmorning()
Traceback (most recent call last):
...
TypeError: goodmorning() takes exactly 1 argument (0 given)
```

Now call the function *goodmorning* with one argument:

```
>>> example1.goodmorning('Jack')
'Good morning, Jack!'
```

See the examples are written like executed from prompt. You can use your python prompt and copy paste from there. Now create another file *test_example1.py* with this content:

```
import unittest
import doctest

def test_suite():
    return unittest.TestSuite((
        doctest.DocFileSuite('example1.txt'),
    ))

if __name__ == '__main__':
    unittest.main(defaultTest='test_suite')
```

This is just boilerplate code for running the test. Now run the test using *python2.4 test_example1.py* command. You will get output with following text:

```
File "example1.txt", line 16, in example1.txt
Failed example:
    example1.goodmorning('Jack')
Expected:
    'Good morning, Jack!'
Got nothing
```

Now one test failed, so implement the function now:

```
def goodmorning(name):  
    "This returns a good morning message"  
    return "Good morning, %s!" % name
```

Now run the test again, it will run without failures.

Now start thinking about other functionalities required for the module. Before start coding write about it in text file. Decide API, write test, write code, than continue this cycle until you finish your requirements.

4.1.1 Running tests

By conventions your test modules are put in tests module under each package. But the doctest files can be placed in the package itself. For example if the package is *ticketcollector*. Then the main doctest file can be placed in *ticketcollector/README.txt*. And create a sub-package *zopetic.tests*, under this package create test modules like *test_main.py*, *test_extra.py* etc.

To run the unit tests, change to instance home:

```
$ cd ticketcollector  
$ ./bin/test
```


Chapter 5

Browser Resources

5.1 File Resource

Certain presentation, like images and style sheets are not associated with any other component, so that one cannot create a view. To solve this problem, resources were developed, which are presentation components that do not require any context. This mini-chapter will demonstrate how resources are created and registered with Zope 3.

The first goal is to register a simple plain-text file called *resource.txt* as a browser resource. The first step is to create this file anywhere you wish on the filesystem, and adding the following content:

```
Hello, I am a Zope 3 Resource Component!
```

Now just register the resource in a ZCML configuration file using the *browser* resource directive:

```
<browser:resource
  name="resource.txt"
  file="resource.txt"
  layer="default" />
```

Line 2: This is the name under which the resource will be known in Zope.

Line 3: The `file` attribute specifies the path to the resource on the filesystem. The current working directory (`'.'`) is always the directory the configuration file is located. So in the example above, the file `resource.txt` is located in the same folder as the configuration file is.

Line 4: The optional `layer` attribute specifies the layer the resource is added to. By default, the default layer is selected.

Once you hook up the configuration file to the main configuration path and restart Zope 3, you should be able to access the resource now via a Browser using `http://localhost:8080/@@/resource.txt`. The `@@/` in the URL tells the traversal mechanism that the following object is a resource.

5.2 Image Resource

If you have an image resource, you might want to use different configuration. Create a simple image called `img.png` and register it as follows:

```
<browser:resource
  name="img.png"
  image="img.png"
  permission="zope.ManageContent" />
```

Line 3: As you can see, instead of the `file` attribute we use the `image` one. Internally this will create an `Image` object, which is able to detect the content type and returns it correctly. There is a third possible attribute named `template`. If specified, a Page Template that is executed when the resource is called. Note that only one of `file`, `image`, or `template` attributes can be specified inside a resource directive.

Line 4: A final optional attribute is the "permission" one must have to view the resource. To demonstrate the security, I set the permission required for viewing the image to `zope.ManageContent`, so that you must log in as an administrator/- manager to be able to view it. The default of the attribute is `zope.Public` so that everyone can see the resource.

5.3 Directory Resource

If you have many resource files to register, it can be very tedious to write a single directive for every resource. For this purpose the *resourceDirectory* is provided, with which you can simply declare an entire directory, including its content as resources. Thereby the filenames of the files are reused as the names for the resource available. Assuming you put your two previous resources in a directory called *resource*, then you can use the following:

```
<browser:resourceDirectory
  name="resources"
  directory="../resource" />
```

The image will then be publically available under the URL: *http://localhost:8080/@@/resources/img.png*

The *DirectoryResource* object uses a simple resource type recognition. It looks at the filename extensions to discover the type. For page templates, currently the extensions "pt", "zpt" and "html" are registered and for an image "gif", "png" and "jpg". All other extensions are converted to file resources. Note that it is not necessary to have a list of all image types, since only Browser-displayable images must be recognized.

5.4 ZRT Resource

When working locally, you may be storing your image resources in a directory. If you have a subfolder called *images* with an image *logo.png*. And you have a template, so here is the HTML to insert the logo:

```

```

Now you can see that the template locally works.

If you view the HTML via Zope, you can see that it is broken.

Now, let's try to register the logo with the system like this:

```
<resource
  name="logo.png"
  file="images/logo.png"
/>
```

Now try again, after restarting Zope 3, you can see that it is still broken!. So, relative path is not correct.

Zope Resource Templates (ZRT) allows for locally working resources to work with Zope 3 as well. It will rewrite text segments in a resource. It is a 3rd party package developed by Stephan Richter for Lovely Systems. The package is available from here: <http://pypi.python.org/pypi/z3c.zrtresource>

Add the following lines to the HTML resource:

```
<!--
  /* zrt-replace: "../images/logo.png" \
                  tal"string:${context/++resource++logo.png}" */
-->
```

Then convert HTML resource registration to:

```
<zrt-resource
  name="helloworld.html"
  file="helloworld.html"
/>
```

Chapter 6

Pages

Chapter 7

Content Components

Chapter 8

Internationalization and Localization

