Introduction

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Introduction

Zope is an open source web application framework. Zope has three distinct audiences: users, developers, and administrators. This guide is intended to document Zope for the second audience, developers. If you are a user, or an administrator, or are looking for documentation about either of those audiences, you should check out The Zope Book or The Zope Administrator's Guide.

Throughout this guide, it is assumed that you know how to program in the Python programming language. Most of the examples in this book will be in Python. There are a number of great resources and books for learning Python; the best online resource is the Python.org web site and many books can be found on the shelves of your local bookstore.

Zope is a large, mature framework that provides many services to web application developers. This book describes these services from a hands on, example−oriented standpoint. This book is not a complete reference to the Zope API, but rather a practical guide to applying Zope's services to develop and deploy your own web applications built on Zope. This book covers the following topics:

Components and Interfaces
: Zope is moving toward a component−centric development model. This chapter describes the new component model in Zope and how Zope components are described through interfaces.

Object Publishing
: Developing applications for Zope involves more than just creating a component, that component must be *publishable* on the web. This chapter describes publication, and how your components need to be designed to be published.

Zope Products
: New Zope components are distributed and installed in packages called *Products*. This chapter explains Products in detail.

Persistent Components
: Zope provides a built−in, transparent Python object database called ZODB. This chapter describes how to create persistent components, and how they work in conjunction with the ZODB.

Acquisition
: Zope relies heavily on a dynamic technique called *acquisition*. This chapter explores acquisition thoroughly.

Security
: When your component is used by many different people through the web, security becomes a big concern. This chapter describes Zope's security API and how you can use it to make security assertions about your object.

Debugging and Testing
: Zope has built in debugging and testing support. This chapter describes these facilities and how you can debug and test your components.

Chapter 1: Components and Interfaces

Zope is becoming a component system. Zope components will be Python objects with interfaces that describe them. Right now only some of the Zope code base uses interfaces. In coming releases more and more of Zope will include interfaces. As a Zope developer you can use interfaces right now to build your Zope components.

Zope Components

Components are objects that are associated with interfaces. An interface is a Python object that describes how you work with other Python objects. In this chapter, you'll see some simple examples of creating components, and a description of interfaces and how they work.

Here is a very simple component that says hello. Like all components, this one generally consists of two peices, an interface, and an implementation:

```
from Interface import Base

class Hello(Base):
    """ The Hello interface provides greetings. """

    def hello(self, name):
        """ Say hello to the name """

class HelloComponent:

    __implements__ = Hello

    def hello(self, name):
        return "hello %s!" % name
```

Let's take a look at this step by step. Here, you see two Python class statements. The first statement creates the *interface*, and the second statement creates the *implementation*.

The first class statement creates the `Hello` interface. This interface describes one method, called `hello`. Notice that there is no implementation for this method, interfaces do not define behavior, they just describe a specification.

The second `class` statement creates the `HelloComponent` class. This class is the actual component that *does* what `Hello` *describes*. This is usualy referred to as the *implementation* of `Hello`. In order for you to know what interfaces `HelloComponent` implements, it must somehow associate itself with an interface. The `__implements__` class attribute does just that. It says, "I implement these interfaces". In this case, `HelloComponent` asserts that it implements one interface, `Hello`.

The interface describes how you would work with the object, but it doesn't dictate how that description is implemented. For example, here's a more complex implementation of the `Hello` interface:

```
import xmlrpclib
class XMLRPCHello:

    __implements__ = Hello

    def hello(self, name):
        """
```

```
            Delegates the hello call to a remote object using XML−RPC.
            """
            s = xmlrpclib.Server('http://www.zope.org/')
            return s.hello(name)
```

This component contacts a remote server and gets its hello greeting from a remote component.

And that's all there is to components, really. The rest of this chapter describes interfaces and how you can work with them from the perspective of components. In Chapter 3, we'll put all this together into a Zope product.

Python Interfaces

Interface describe the behavior of an object by containing useful information about the object. This information includes:

- Prose documentation about the object. In Python terms, this is called the "doc string" of the interface. In this element, you describe how the object works in prose language and any other useful information about the object.
- Descriptions of attributes. Attribute descriptions include the name of the attribute and prose documentation describing the attributes usage.
- Descriptions of methods. Method descriptions can include:
    - ♦ Prose "doc string" documentation about the method and its usage.
    - ♦ A sequence of parameter objects that describes the parameters expected by the method.
- Optional tagged data. Interface objects (and their attributes, methods, and method parameters) can have optional, application specific tagged data associated with them. Examples uses for this are security assertions, pre/post conditions, unit tests, and other possible information you may want to associate with an Interface or its attributes.

Not all of this information is mandatory. For example, you may only want the methods of your interface to have prose documentation and not describe the arguments of the method in exact detail. Interface objects are flexible and let you give or take any of these components.

Why Use Interfaces?

Interfaces solve a number of problems that arise while developing large systems with lots of developers.

- Developers waste a lot of time looking at the source code of your system to figure out how objects work. This is even worse if someone else has already wasted their time doing the same thing.
- Developers who are new to your system may misunderstand how your object works, causing, and possibly propagating, usage errors.
- Because an object's interface is inferred from the source, developers may end up using methods and attributes that are meant for "internal use only".
- Code inspection can be hard, and very discouraging to novice programmers trying to understand code written by gurus.

Interfaces try to solve these problems by providing a way for you to describe how to use an object, and a mechanism for discovering that description.

Creating Interfaces

The first step to creating a component, as you've been shown, is to create an interface.

Interface objects can be conveniently constructed using the Python `class` statement. Keep in mind that this syntax can be a little misleading, because interfaces are *not* classes. It is important to understand that using Python's class syntax is just a convenience, and that the resulting object is an *interface*, not a class.

To create an interface object using Python's class syntax, create a Python class that subclasses from `Interface.Base`:

```
from Interface import Base

class Hello(Base):

    def hello(self, name):
        """ Say hello to the world """
```

This interface does not implement behavior for its methods, it just describes an interface that a typical "Hello" object would realize. By subclassing the `Interface.Base` interface, the resulting object `Hello` is an interface object. The Python interpreter confirms this:

```
>>> Hello
<Interface Hello at 812cbd4>
```

Now, you can associate the `Hello` Interface with your new, concrete class in which you define your user behavior. For example:

```
class HelloComponent:

    __implements__ = Hello

    def hello(self, name):
        return "Hello %s!" % name
```

This new class, `HelloComponent` is a concrete class that implements the `Hello` interface. A class can realize more than one interface. For example, say you had an interface called `Item` that described how an object worked as an item in a "Container" object. If you wanted to assert that `HelloComponent` instances realized the `Item` interface as well as `Hello`, you can provide a sequence of Interface objects to the `HelloComponent` class:

```
class HelloComponent:

    __implements__ = Hello, Item
```

This `__implements__` attribute is called an *interface assertion*. An interface assertion can be either an interface, or a sequence of interface assertions. Here's a more complex example:

```
class Sandwich:

    __implements__ = (Food, (Nourishing, Delicious), (GetsStaleQuickly,
                     (EdibleWithHands, GoodForLunch)))
```

Interface assertions allow complex nesting of interfaces. This is mostly useful when you wish to assert that your class implements some specific interfaces, along with whatever interfaces your base class implements:

```
class Sandwhich(Food):

    __implements__ = (EdibleWithHands, GoodForLunch, Food.__implements__)
```

Take care before you assert that your class implements the interfaces of your base classes.

The Interface Model

Interfaces can extend other interfaces. For example, let's extend the `Hello` interface by adding an additional method:

```
class SmartHello(Hello):
    """  A Hello object that remembers who it's greeted """

    def lastGreeted(self):
        """ Returns the name of the last person greeted. """
```

`SmartHello` extends the `Hello` interface. It does this by using the same syntax a class would use to subclass another class.

Now, you can ask the `SmartHello` for a list of the interfaces it extends with `getBases`:

```
>>> SmartHello.getBases()
[<interface Hello at 80c72c8>]
```

An interface can extend any number of other interfaces, and `getBases` will return that list of interfaces for you. If you want to know if `SmartHello` extends any other interface, you could call `getBases` and search through the list, but a convenience method called `extends` is provided that returns true or false for this purpose:

```
>>> SmartHello.extends(Hello)
1
>>> SmartHello.extends(Sandwich)
0
>>>
```

Here you can see `extends` can be used to determine if one interface extends another.

You may notice a similarity between interfaces extending from other interfaces and classes sub−classing from other classes. This *is* a similar concept, but the two should not be considered equal. There is no assumption that classes and interfaces exist in a one to one relationship; one class may implement several interfaces, and a class may not implement its base classes's interfaces.

The distinction between a class and an interface should always be kept clear. The purpose of a class is to share the implementation of how an object works. The purpose of an interface is to document how to work *with* an object, not how the object is implemented. It is possible to have several different classes with very different implementations realize the same interface. Because of this, interfaces and classes should never be confused.

Querying an Interface

Interfaces can be queried for information. The simplest case is to ask an interface the names of all the various interface items it describes. From the Python interpreter, for example, you can walk right up to an interface and ask it for its *names*:

```
>>> User.names()
['getUserName', 'getFavoriteColor', 'getPassword']
```

Interfaces can also give you more interesting information about their items. Interface objects can return a list of `(name, description)` tuples about their items by calling the *namesAndDescriptions* method. For example:

```
>>> User.namesAndDescriptions()
[('getUserName', <Interface.Method.Method instance at 80f38f0>),
('getFavoriteColor', <Interface.Method.Method instance at 80b24f0>),
('getPassword', <Interface.Method.Method instance at 80fded8>)]
```

As you can see, the "description" of the Interface's three items in these cases are all `Method` objects. Descriptions objects can be either `Attribute` or `Method` objects. Attributes, methods, and interface objects implement the following interface:

```
'getName()' -- Returns the name of the object.

'getDoc()' -- Returns the documentation for the object.
```

Method objects provide a way to describe rich meta−data about Python methods. Method objects have the following methods:

`getSignatureInfo()`
: Returns a dictionary describing the method parameters.

`getSignatureString()`
: Returns a human−readable string representation of the method's signature.

For example:

```
>>> m=User.namesAndDescriptions()[0][1]
>>> m
<Interface.Method.Method instance at 80f38f0>
>>> m.getSignatureString()
'(fullName=1)'
>>> m.getSignatureInfo()
{'varargs': None, 'kwargs': None, 'optional': {'fullName': 1},
'required': (), 'positional': ('fullName',)}
```

You can use `getSignatureInfo` to find out the names and types of the method parameters.

Checking Implementation

You can ask an interface if a certain class or instance that you hand it implements that interface. For example, say you want to know if instances of the `HelloComponent` class implement `Hello`:

```
Hello.implementedByInstancesOf(HelloComponent)
```

This is a true expression. If you had an instance of `HelloComponent`, you can also ask the interface if that instance implements the interface:

```
Hello.implementedBy(my_hello_instance)
```

This would also return true if *my_hello_instance* was an instance of *HelloComponent*, or any other class that implemented the *Hello* Interface.

Conclusion

Interfaces provide a simple way to describe your Python objects. By using interfaces you document your objects' capabilities. As Zope becomes more component oriented, your objects will fit right in. While components and interfaces are forward looking technologies, they are useful today for documentation and verification.

Chapter 2: Object Publishing

Introduction

Zope puts your objects on the web. This is called *object publishing*. One of Zope's unique characteristics is the way it allows you to walk up to your objects and call methods on them with simple URLs. In addition to HTTP, Zope makes your objects available to other network protocols including FTP, WebDAV and XML−RPC.

In this chapter you'll find out exactly how Zope publishes objects. You'll learn all you need to know in order to design your objects for web publishing.

HTTP Publishing

When you contact Zope with a web browser, your browser sends an HTTP request to Zope's web server. After the request is completely received, it is processed by `ZPublisher`, which is Zope's object publisher. `ZPublisher` is a kind of light−weight ORB (Object Request Broker). It takes the request and locates an object to handle the request. The publisher uses the request URL as a map to locate the published object. Finding an object to handle the request is called *traversal*, since the publisher moves from object to object as it looks for the right one. Once the published object is found, the publisher calls a method on the published object, passing it parameters as necessary. The publisher uses information in the request to determine which method to call, and what parameters to pass. The process of extracting parameters from the request is called *argument marshalling*. The published object then returns a response, which is passed back to Zope's web server. The web server, then passes the response back to your web browser.

The publishing process is summarized in Figure 2−1

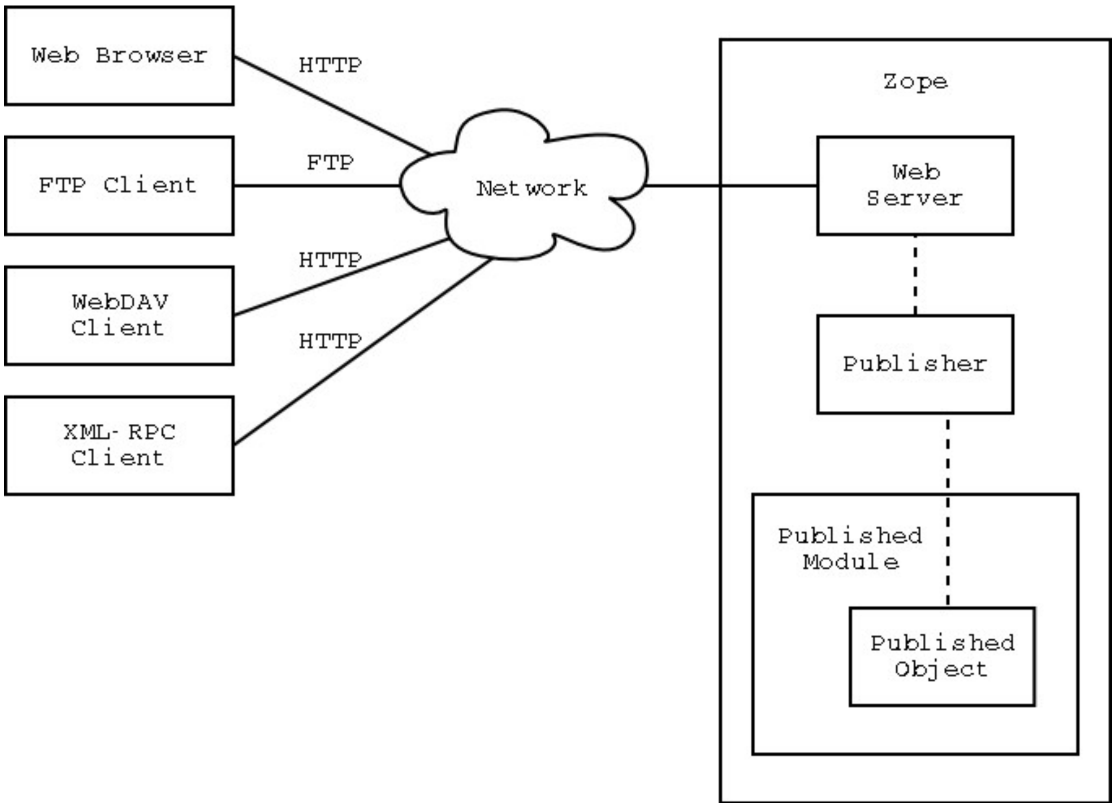

Figure 2−1 Object publishing

Notice how the published object is located within the ZODB, while the publisher and web server are not. See Chapter 4 for more information on the ZODB.

This chapter will cover all the steps of object publishing in detail. To summarize, object publishing consists of the main steps:

1. The client sends a request to the publisher
2. The publisher locates the published object using the request URL as a map.
3. The publisher calls the published object with arguments from the request.
4. The publisher interprets and returns the results to the client.

The chapter will also cover all the technical details, special cases and extra−steps that this list glosses over.

URL Traversal

Traversal is the process the publisher uses to locate the published object. Typically the publisher locates the published object by walking along the URL. Take for example a collection of objects:

```
class Classification:
    ...

class Animal:
    ...

    def screech(self, ...):
        ...
```

```
vertebrates=Classification(...)
vertebrates.mammals=Classification(...)
vertebrates.reptiles=Classification(...)
vertebrates.mammals.monkey=Animal(...)
vertebrates.mammals.dog=Animal(...)
vertebrates.reptiles.lizard=Animal(...)
```

This collection of objects forms an object hierarchy. Using Zope you can publish objects with URLs. For example, the URL `http://zope/vertebrates/mammals/monkey/screech`, will traverse the object hierarchy, find the `monkey` object and call its `screech` method.

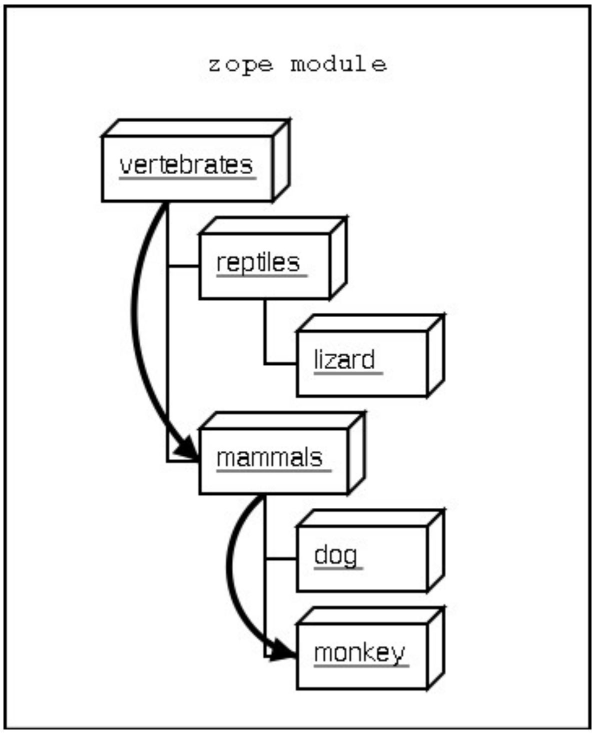

Figure 2−2 Traversal path through an object hierarchy

The publisher starts from the root object and takes each step in the URL as a key to locate the next object. It moves to the next object and continues to move from object to object using the URL as a guide.

Typically the next object is a sub−object of the current object that is named by the path segment. So in the example above, when the publisher gets to the `vertebrates` object, the next path segment is "mammals", and this tells the publisher to look for a sub−object of the current object with that name. Traversal stops when Zope comes to the end of the URL. If the final object is found, then it is published, otherwise an error is returned.

Now let's take a more rigorous look at traversal.

Traversal Interfaces

Zope defines interfaces for publishable objects, and publishable modules.

When you are developing for Zope you almost always use the `Zope` package as your published module.

However, if you are using `ZPublisher` outside of Zope you'll be interested in the published module interface.

Publishable Object Requirements

Zope has few restrictions on publishable objects. The basic rule is that the object must have a doc string. This requirement goes for method objects too.

Another requirement is that a publishable object must not have a name that begin with an underscore. These two restrictions are designed to keep private objects from being published.

Finally, published objects cannot be Python module objects.

Traversal Methods

During traversal, `ZPublisher` cuts the URL into path elements delimited by slashes, and uses each path element to traverse from the current object to the next object. `ZPublisher` locates the next object in one of three ways:

1. Using `__bobo_traverse__`
2. Using `getattr`
3. Using dictionary access.

First the publisher attempts to call the traversal hook method, `__bobo_traverse__`. If the current object has this method it is called with the request and the current path element. The method should return the next object or `None` to indicate that a next object can't be found. You can also return a tuple of objects from `__bobo_traverse__` indicating a sequence of sub−objects. This allows you to add additional parent objects into the request. This is almost never necessary.

Here's an example of how to use `__bobo_traverse__`:

```
def __bobo_traverse__(self, request, key):
    # if there is a special cookie set, return special
    # subobjects, otherwise return normal subobjects

    if request.cookies.has_key('special'):
        # return a subobject from the special dict
        return self.special_subobjects.get(key, None)

    # otherwise return a subobject from the normal dict
    return self.normal_subobjects.get(key, None)
```

This example shows how you can examine the request during the traversal process.

If the current object does not define a `__bobo_traverse__` method, then the next object is searched for using `getattr`. This locates sub−objects in the normal Python sense.

If the next object can't be found with `getattr`, `ZPublisher` calls on the current object as though it were a dictionary. Note: the path element will be a string, not an integer, so you cannot traverse sequences using index numbers in the URL.

For example, suppose `a` is the current object, and `next` is the name of the path element. Here are the three things that `ZPublisher` will try in order to find the next object:

1. `a.__bobo_traverse__("next")`
2. `a.next`
3. `a["next"]`

If the next object isn't found by any of these means `ZPublisher` returns a HTTP 404 (Not Found) exception.

Publishing Methods

Once the published object is located with traversal, Zope *publishes* it in one of three possible ways.

Calling the published object
: If the published object is a function or method or other callable object, the publisher calls it. Later in the chapter you'll find out how the publisher figures out what arguments to pass when calling.

Calling the default method
: If the published object is not callable, the publisher uses the default method. For HTTP `GET` and `POST` requests the default method is `index_html`. For other HTTP requests such as `PUT` the publisher looks for a method named by the HTTP method. So for an HTTP `HEAD` request, the publisher would call the `HEAD` method on the published object.

Stringifying the published object
: If the published object isn't callable, and doesn't have a default method, the publisher publishes it using the Python `str` function to turn it into a string.

After the response method has been determined and called, the publisher must interpret the results.

HTTP Responses

Normally the published method returns a string which is considered the body of the HTTP response. The response headers can be controlled by calling methods on the response object, which is described later in the chapter. Optionally, the published method can return a tuple with the title, and body of the response. In this case, the publisher returns an generated HTML page, with the first item of the tuple used for the HTML `title` of the page, and the second item as the contents of the HTML `body` tag. For example a response of:

```
('response', 'the response')
```

is turned into this HTML page:

```
<html>
<head><title>response</title></head>
<body>the response</body>
</html>
```

Controlling Base HREF

When you publish an object that returns HTML relative links should allow you to navigate between methods. Consider this example:

```
class Example:
    "example"

    def one(self):
        "method one"
        return """<html>
                  <head>
```

```
                          <title>one</title>
                          </head>
                          <body>
                          <a href="two">two</a>
                          </body>
                          </html>"""

            def two(self):
                "method two"
                return """<html>
                          <head>
                          <title>two</title>
                          </head>
                          <body>
                          <a href="one">one</a>
                          </body>
                          </html>"""
```

The relative links in methods `one` and `two` allow you to navigate between the methods.

However, the default method, `index_html` presents a problem. Since you can access the `index_html` method without specifying the method name in the URL, relative links returned by the `index_html` method won't work right. For example:

```
        class Example:
            "example"

             def index_html(self):
                return """<html>
                          <head>
                          <title>one</title>
                          </head>
                          <body>
                          <a href="one">one</a><br>
                          <a href="two">two</a>
                          </body>
                          </html>"""
             ...
```

If you publish an instance of the `Example` class with the URL `http://zope/example`, then the relative link to method `one` will be `http://zope/one`, instead of the correct link, `http://zope/example/one`.

Zope solves this problem for you by inserting a `base` tag inside the `head` tag in the HTML output of `index_html` method when it is accessed as the default method. You will probably never notice this, but if you see a mysterious `base` tag in your HTML output, know you know where it came from. You can avoid this behavior by manually setting your own base with a `base` tag in your `index_html` method output.

Response Headers

The publisher and the web server take care of setting response headers such as `Content-Length` and `Content-Type`. Later in the chapter you'll find out how to control these headers. Later you'll also find out how exceptions are used to set the HTTP response code.

Pre−Traversal Hook

The pre−traversal hook allows your objects to take special action before they are traversed. This is useful for doing things like changing the request. Applications of this include special authentication controls, and virtual hosting support.

If your object has a method named `__before_publishing_traverse__`, the publisher will call it with the current object and the request, before traversing your object. Most often your method will change the request. The publisher ignores anything you return from the pre−traversal hook method.

The `ZPublisher.BeforeTraverse` module contains some functions that help you register pre−traversal callbacks. This allows you to perform fairly complex callbacks to multiple objects when a given object is about to be traversed.

Traversal and Acquisition

Acquisition affects traversal in several ways. See Chapter 5, "Acquisition" for more information on acquisition. The most obvious way in which acquisition affects traversal is in locating the next object in a path. As we discussed earlier, the next object during traversal is often found using `getattr`. Since acquisition affects `getattr`, it will affect traversal. The upshot is that when you are traversing objects that support implicit acquisition, you can use traversal to walk over acquired objects. Consider the object hierarchy rooted in `fruit`:

```
from Acquisition import Implicit

class Node(Implicit):
    ...

fruit=Node()
fruit.apple=Node()
fruit.orange=Node()
fruit.apple.strawberry=Node()
fruit.orange.banana=Node()
```

When publishing these objects, acquisition can come into play. For example, consider the URL */fruit/apple/orange*. The publisher would traverse from `fruit`, to `apple`, and then using acquisition, it would traverse to `orange`.

Mixing acquisition and traversal can get complex. Consider the URL */fruit/apple/orange/strawberry/banana*. This URL is functional but confusing. Here's an even more perverse but legal URL */fruit/apple/orange/orange/apple/apple/banana*.

In general you should limit yourself to constructing URLs which use acquisition to acquire along containment, rather than context lines. It's reasonable to publish an object or method that you acquire from your container, but it's probably a bad idea to publish an object or method that your acquire from outside your container. For example:

```
from Acquisition import Implicit

class Basket(Implicit):
    ...
    def numberOfItems(self):
        "Returns the number of contained items"
        ...
```

```
class Vegetable(Implicit):
    ...
    def texture(self):
        "Returns the texture of the vegetable."

class Fruit(Implicit):
    ...
    def color(self):
        "Returns the color of the fruit."

 basket=Basket()
 basket.apple=Fruit()
 basket.carrot=Vegetable()
```

The URL */basket/apple/numberOfItems* uses acquisition along containment lines to publish the `numberOfItems` method (assuming that `apple` doesn't have a `numberOfItems` attribute). However, the URL */basket/carrot/apple/texture* uses acquisition to locate the `texture` method from the `apple` object's context, rather than from its container. While this distinction may be obscure, the guiding idea is to keep URLs as simple as possible. By keeping acquisition simple and along containment lines your application increases in clarity, and decreases in fragility.

A second usage of acquisition in traversal concerns the request. The publisher tries to make the request available to the published object via acquisition. It does this by wrapping the first object in an acquisition wrapper that allows it to acquire the request with the name `REQUEST`. This means that you can normally acquire the request in the published object like so:

```
request=self.REQUEST # for implicit acquirers
```

or like so:

```
request=self.aq_acquire('REQUEST') # for explicit acquirers
```

Of course, this will not work if your objects do not support acquisition, or if any traversed objects have an attribute named `REQUEST`.

Finally, acquisition has a totally different role in object publishing related to security which we'll examine next.

Traversal and Security

As the publisher moves from object to object during traversal it makes security checks. The current user must be authorized to access each object along the traversal path. The publisher controls access in a number of ways. For more information about Zope security, see Chapter 6, "Security".

Basic Publisher Security

The publisher imposes a few basic restrictions on traversable objects. These restrictions are the same of those for publishable objects. As previously stated, publishable objects must have doc strings and must not have names beginning with underscore.

The following details are not important if you are using the Zope framework. However, if your are publishing your own modules, the rest of this section will be helpful.

The publisher checks authorization by examining the `__roles__` attribute of each object as it performs traversal. If present, the `__roles__` attribute should be `None` or a list of role names. If it is None, the object is considered public. Otherwise the access to the object requires validation.

Some objects such as functions and methods do not support creating attributes (at least they didn't before Python 2). Consequently, if the object has no `__roles__` attribute, the publisher will look for an attribute on the object's parent with the name of the object followed by `__roles__`. For example, a function named `getInfo` would store its roles in its parent's `getInfo__roles__` attribute.

If an object has a `__roles__` attribute that is not empty and not `None`, the publisher tries to find a user database to authenticate the user. It searches for user databases by looking for an `__allow_groups__` attribute, first in the published object, then in the previously traversed object, and so on until a user database is found.

When a user database is found, the publisher attempts to validate the user against the user database. If validation fails, then the publisher will continue searching for user databases until the user can be validated or until no more user databases can be found.

The user database may be an object that provides a validate method:

```
validate(request, http_authorization, roles)
```

where `request` is a mapping object that contains request information, `http_authorization` is the value of the HTTP `Authorization` header or `None` if no authorization header was provided, and `roles` is a list of user role names.

The validate method returns a user object if succeeds, and `None` if it cannot validate the user. See Chapter 6 for more information on user objects. Normally, if the validate method returns `None`, the publisher will try to use other user databases, however, a user database can prevent this by raising an exception.

If validation fails, Zope will return an HTTP header that causes your browser to display a user name and password dialog. You can control the realm name used for basic authentication by providing a module variable named `__bobo_realm__`. Most web browsers display the realm name in the user name and password dialog box.

If validation succeeds the publisher assigns the user object to the request variable, `AUTHENTICATED_USER`. The publisher places no restriction on user objects.

Zope Security

When using Zope rather than publishing your own modules, the publisher uses acquisition to locate user folders and perform security checks. The upshot of this is that your published objects must inherit from `Acquisition.Implicit` or `Acquisition.Explicit`. See Chapter 5, "Acquisition", for more information about these classes. Also when traversing each object must be returned in an acquisition context. This is done automatically when traversing via `getattr`, but you must wrap traversed objects manually when using `__getitem__` and `__bobo_traverse__`. For example:

```
class Example(Acquisition.Explicit):
    ...

    def __bobo_traverse__(self, name, request):
        ...
```

```
            next_object=self._get_next_object(name)
            return  next_object.__of__(self)
```

Additionally you will need to make security declarations on your traversed object using `ClassSecurityInfo` as described in Chapter 6, "Security".

Finally, traversal security can be circumvented with the `__allow_access_to_unprotected_subobjects__` attribute as described in Chapter 6, "Security".

Environment Variables

You can control some facets of the publisher's operation by setting environment variables.

`Z_DEBUG_MODE`
: Sets debug mode. In debug mode tracebacks are not hidden in error pages. Also debug mode causes `DTMLFile` objects, External Methods and help topics to reload their contents from disk when changed. You can also set debug mode with the `-D` switch when starting Zope.

`Z_REALM`
: Sets the basic authorization realm. This controls the realm name as it appears in the web browser's username and password dialog. You can also set the realm with the `__bobo_realm__` module variable, as mentioned previously.

`PROFILE_PUBLISHER`
: Turns on profiling and sets the name of the profile file. See the Python documentation for more information about the Python profiler.

Many more options can be set using switches on the startup script. See the *Zope Administrator's Guide* for more information.

Testing

ZPublisher comes with built−in support for testing and working with the Python debugger. This topic is covered in more detail in Chapter 7, "Testing and Debugging".

Publishable Module

If you are using the Zope framework, this section will be irrelevant to you. However, if you are publishing your own modules with `ZPublisher` read on.

The publisher begins the traversal process by locating an object in the module's global namespace that corresponds to the first element of the path. Alternately the first object can be located by one of two hooks.

If the module defines a `web_objects` or `bobo_application` object, the first object is searched for in those objects. The search happens according to the normal rules of traversal, using `__bobo_traverse__`, `getattr`, and `__getitem__`.

The module can receive callbacks before and after traversal. If the module defines a `__bobo_before__` object, it will be called with no arguments before traversal. Its return value is ignored. Likewise, if the module defines a `__bobo_after__` object, it will be called after traversal with no arguments. These callbacks can be used for things like acquiring and releasing locks.

Calling the Published Object

Now that we've covered how the publisher located the published object and what it does with the results of calling it, let's take a closer look at how the published object is called.

The publisher marshals arguments from the request and automatically makes them available to the published object. This allows you to accept parameters from web forms without having to parse the forms. Your objects usually don't have to do anything special to be called from the web. Consider this function:

```
def greet(name):
    "greet someone"
    return "Hello, %s" % name
```

You can provide the `name` argument to this function by calling it with a URL like *greet?name=World*. You can also call it with a HTTP `POST` request which includes `name` as a form variable.

In the next sections we'll take a closer look at how the publisher marshals arguments.

Marshalling Arguments from the Request

The publisher marshals form data from GET and POST requests. Simple form fields are made available as Python strings. Multiple fields such as form check boxes and multiple selection lists become sequences of strings. File upload fields are represented with `FileUpload` objects. File upload objects behave like normal Python file objects and additionally have a `filename` attribute which is the name of the file and a `headers` attribute which is a dictionary of file upload headers.

The publisher also marshals arguments from CGI environment variables and cookies. When locating arguments, the publisher first looks in CGI environment variables, then other request variables, then form data, and finally cookies. Once a variable is found, no further searching is done. So for example, if your published object expects to be called with a form variable named `SERVER_URL`, it will fail, since this argument will be marshaled from the CGI environment first, before the form data.

The publisher provides a number of additional special variables such as `URL0` which are derived from the request. These are covered in the `HTTPRequest` API documentation.

Argument Conversion

The publisher supports argument conversion. For example consider this function:

```
def onethird(number):
    "returns the number divided by three"
    return number / 3.0
```

This function cannot be called from the web because by default the publisher marshals arguments into strings, not numbers. This is why the publisher provides a number of converters. To signal an argument conversion you name your form variables with a colon followed by a type conversion code. For example, to call the above function with 66 as the argument you can use this URL *onethird?number:int=66* The publisher supports many converters:

float

Python floating point numbers.

int
: Python integers.

long
: Python long integers.

string
: Python strings.

required
: Non−blank Python strings. Raises an exception if the request does not contain the variable.

date
: Date−time values.

list
: Python list of values, even if there is only one value.

lines
: Python list of values obtained by splitting the variable on line breaks.

tokens
: Python list of values obtained by splitting the variable on spaces.

tuple
: Python tuple of values, even if there is only one.

If the publisher cannot coerce a request variable into the type required by the type converter it will raise an error. This is useful for simple applications, but restricts your ability to tailor error messages. If you wish to provide your own error messages, you should convert arguments manually in your published objects rather than relying on the publisher for coercion. Another possibility is to use JavaScript to validate input on the client−side before it is submitted to the server.

You can combine type converters to a limited extent. For example you could create a list of integers like so:

```
<input type="checkbox" name="numbers:list:int" value="1">
<input type="checkbox" name="numbers:list:int" value="2">
<input type="checkbox" name="numbers:list:int" value="3">
```

In addition to these type converters, the publisher also supports method and record arguments.

Method Arguments

Sometimes you may wish to control which object is published based on form data. For example, you might want to have a form with a select list that calls different methods depending on the item chosen. Similarly, you might want to have multiple submit buttons which invoke a different method for each button.

The publisher provides a way to select methods using form variables through use of the *method* argument type. The method type allows the request `PATH_INFO` to be augmented using information from a form item name or value.

If the name of a form field is `:method`, then the value of the field is added to `PATH_INFO`. For example, if the original `PATH_INFO` is `foo/bar` and the value of a `:method` field is `x/y`, then `PATH_INFO` is transformed to `foo/bar/x/y`. This is useful when presenting a select list. Method names can be placed in the select option values.

If the name of a form field ends in `:method` and is longer than 7 characters, then the part of the name before `:method` is added to `PATH_INFO`. For example, if the original `PATH_INFO` is `foo/bar` and there is a `x/y:method` field, then `PATH_INFO` is transformed to `foo/bar/x/y`. In this case, the form value is ignored. This is useful for mapping submit buttons to methods, since submit button values are displayed and should, therefore, not contain method names.

Only one method field should be provided. If more than one method field is included in the request, the behavior is undefined.

Record Arguments

Sometimes you may wish to consolidate form data into a structure rather than pass arguments individually. Record arguments allow you to do this.

The `record` type converter allows you to combine multiple form variables into a single input variable. For example:

```
<input name="date.year:record:int">
<input name="date.month:record:int">
<input name="date.day:record:int">
```

This form will result in a single variable, `date`, with attributes `year`, `month`, and `day`.

You can skip empty record elements with the `ignore_empty` converter. For example:

```
<input type="text" name="person.email:record:ignore_empty">
```

When the email form field is left blank the publisher skips over the variable rather than returning a null string as its value. When the record `person` is returned it will not have an `email` attribute if the user did not enter one.

You can also provide default values for record elements with the `default` converter. For example:

```
<input type="hidden"
       name="pizza.toppings:record:list:default"
       value="All">
<select multiple name="pizza.toppings:record:list:ignore_empty">
<option>Cheese</option>
<option>Onions</option>
<option>Anchovies</option>
<option>Olives</option>
<option>Garlic<option>
</select>
```

The `default` type allows a specified value to be inserted when the form field is left blank. In the above example, if the user does not select values from the list of toppings, the default value will be used. The record `pizza` will have the attribute `toppings` and its value will be the list containing the word "All" (if the field is empty) or a list containing the selected toppings.

You can even marshal large amounts of form data into multiple records with the `records` type converter. Here's an example:

```
<h2>Member One</h2>
Name:
<input type="text" name="members.name:records"><BR>
Email:
<input type="text" NAME="members.email:records"><BR>
Age:
```

```
<input type="text" NAME="members.age:int:records"><BR>

<H2>Member Two</H2>
Name:
<input type="text" name="members.name:records"><BR>
Email:
<input type="text" NAME="members.email:records"><BR>
Age:
<input type="text" NAME="members.age:int:records"><BR>
```

This form data will be marshaled into a list of records named `members`. Each record will have a `name`, `email`, and `age` attribute.

Record marshalling provides you with the ability to create complex forms. However, it is a good idea to keep your web interfaces as simple as possible.

Exceptions

Unhandled exceptions are caught by the object publisher and are translated automatically to nicely formatted HTTP output.

When an exception is raised, the exception type is mapped to an HTTP code by matching the value of the exception type with a list of standard HTTP status names. Any exception types that do not match standard HTTP status names are mapped to "Internal Error" (500). The standard HTTP status names are: "OK", "Created", "Accepted", "No Content", "Multiple Choices", "Redirect", "Moved Permanently", "Moved Temporarily", "Not Modified", "Bad Request", "Unauthorized", "Forbidden", "Not Found", "Internal Error", "Not Implemented", "Bad Gateway", and "Service Unavailable". Variations on these names with different cases and without spaces are also valid.

An attempt is made to use the exception value as the body of the returned response. The object publisher will examine the exception value. If the value is a string that contains some white space, then it will be used as the body of the return error message. If it appears to be HTML, the error content type will be set to `text/html`, otherwise, it will be set to `text/plain`. If the exception value is not a string containing white space, then the object publisher will generate its own error message.

There are two exceptions to the above rule:

1. If the exception type is: "Redirect", "Multiple Choices" "Moved Permanently", "Moved Temporarily", or "Not Modified", and the exception value is an absolute URI, then no body will be provided and a `Location` header will be included in the output with the given URI.
2. If the exception type is "No Content", then no body will be returned.

When a body is returned, traceback information will be included in a comment in the output. As mentioned earlier, the environment variable `Z_DEBUG_MODE` can be used to control how tracebacks are included. If this variable is set then tracebacks are included in `PRE` tags, rather than in comments. This is very handy during debugging.

Exceptions and Transactions

When Zope receives a request it begins a transaction. Then it begins the process of traversal. Zope automatically commits the transaction after the published object is found and called. So normally each web request constitutes one transaction which Zope takes care of for you. See Chapter 4. for more information on transactions.

If an unhandled exception is raised during the publishing process, Zope aborts the transaction. As detailed in Chapter 4. Zope handles `ConflictErrors` by re−trying the request up to three times. This is done with the `zpublisher_exception_hook`.

In addition, the error hook is used to return an error message to the user. In Zope the error hook creates error messages by calling the `raise_standardErrorMessage` method. This method is implemented by `SimpleItem.Item`. It acquires the `standard_error_message` DTML object, and calls it with information about the exception.

You will almost never need to override the `raise_standardErrorMessage` method in your own classes, since it is only needed to handle errors that are raised by other components. For most errors, you can simply catch the exceptions normally in your code and log error messages as needed. If you need to, you should be able to customize application error reporting by overriding the `standard_error_message` DTML object in your application.

Manual Access to Request and Response

You do not need to access the request and response directly most of the time. In fact, it is a major design goal of the publisher that most of the time your objects need not even be aware that they are being published on the web. However, you have the ability to exert more precise control over reading the request and returning the response.

Normally published objects access the request and response by listing them in the signature of the published method. If this is not possible you can usually use acquisition to get a reference to the request. Once you have the request, you can always get the response from the request like so:

```
response=REQUEST.RESPONSE
```

The APIs of the request and response are covered in the API documentation. Here we'll look at a few common uses of the request and response.

One reason to access the request is to get more precise information about form data. As we mentioned earlier, argument marshalling comes from a number of places including cookies, form data, and the CGI environment. For example, you can use the request to differentiate between form and cookie data:

```
cookies=REQUEST.cookies # a dictionary of cookie data
form=REQUEST.form # a dictionary of form data
```

One common use of the response object is to set response headers. Normally the publisher in concert with the web server will take care of response headers for you. However, sometimes you may wish manually control headers:

```
RESPONSE.setHeader('Pragma', 'No−Cache')
```

Another reason to access the response is to stream response data. You can do this with the `write` method:

```
while 1:
    data=getMoreData() #this call may block for a while
    if not data:
        break
        RESPONSE.write(data)
```

Here's a final example that shows how to detect if your method is being called from the web. Consider this function:

```
def feedParrot(parrot_id, REQUEST=None):
    ...

    if REQUEST is not None:
        return "<html><p>Parrot %s fed</p></html>" % parrot_id
```

The `feedParrot` function can be called from Python, and also from the web. By including `REQUEST=None` in the signature you can differentiate between being called from Python and being called form the web. When the function is called from Python nothing is returned, but when it is called from the web the function returns an HTML confirmation message.

Other Network Protocols

FTP

Zope comes with an FTP server which allows users to treat the Zope object hierarchy like a file server. As covered in Chapter 3, Zope comes with base classes ('SimpleItem' and 'ObjectManager') which provide simple FTP support for all Zope objects. The FTP API is covered in the API reference.

To support FTP in your objects you'll need to find a way to represent your object's state as a file. This is not possible or reasonable for all types of objects. You should also consider what users will do with your objects once they access them via FTP. You should find out which tools users are likely to edit your object files. For example, XML may provide a good way to represent your object's state, but it may not be easily editable by your users. Here's an example class that represents itself as a file using RFC 822 format:

```
from rfc822 import Message
from cStringIO import StringIO

class Person(...):

    def __init__(self, name, email, age):
        self.name=name
        self.email=email
        self.age=age

    def writeState(self):
        "Returns object state as a string"
        return "Name: %s\nEmail: %s\nAge: %s" % (self.name,
                                                 self.email,
                                                 self.age)
    def readState(self, data):
        "Sets object state given a string"
        m=Message(StringIO(data))
        self.name=m['name']
        self.email=m['email']
        self.age=int(m['age'])
```

The `writeState` and `readState` methods serialize and unserialize the `name`, `age`, and `email` attributes to and from a string. There are more efficient ways besides RFC 822 to store instance attributes in a file, however RFC 822 is a simple format for users to edit with text editors.

To support FTP all you need to do at this point is implement the `manage_FTPget` and `PUT` methods. For example:

```
        def manage_FTPget(self):
            "Returns state for FTP"
            return self.writeState()

        def PUT(self, REQUEST):
            "Sets state from FTP"
             self.readState(REQUEST['BODY'])
```

You may also choose to implement a `get_size` method which returns the size of the string returned by `manage_FTPget`. This is only necessary if calling `manage_FTPget` is expensive, and there is a more efficient way to get the size of the file. In the case of this example, there is no reason to implement a `get_size` method.

One side effect of implementing `PUT` is that your object now supports HTTP PUT publishing. See the next section on WebDAV for more information on HTTP PUT.

That's all there is to making your object work with FTP. As you'll see next WebDAV support is similar.

WebDAV

WebDAV is a protocol for collaboratively edit and manage files on remote servers. It provides much the same functionality as FTP, but it works over HTTP.

It is not difficult to implement WebDAV support for your objects. Like FTP, the most difficult part is to figure out how to represent your objects as files.

Your class must inherit from `webdav.Resource` to get basic DAV support. However, since `SimpleItem` inherits from `Resource`, your class probably already inherits from `Resource`. For container classes you must inherit from `webdav.Collection`. However, since `ObjectManager` inherits from `Collection` you are already set so long as you inherit from `ObjectManager`.

In addition to inheriting from basic DAV classes, your classes must implement `PUT` and `manage_FTPget`. These two methods are also required for FTP support. So by implementing WebDAV support, you also implement FTP support.

The permissions that you assign to these two methods will control the ability to read and write to your class through WebDAV, but the ability to see your objects is controlled through the "WebDAV access" permission.

Supporting Write Locking

Write locking is a feature of WebDAV that allows users to put a lock on objects they are working on. Support write locking is easy. To implement write locking you must assert that your class implements the `WriteLockInterface`. For example:

```
          from webdav.WriteLockInterface import WriteLockInterface

          class MyContentClass(OFS.SimpleItem.Item, Persistent):
              __implements__ = (WriteLockInterface,)
```

It's sufficient to inherit from `SimpleItem.Item`, since it inherits from `webdav.Resource`, which provides write locking along with other DAV support.

In addition, your `PUT` method should begin with calls to `dav__init` and `dav_simpleifhandler`. For example:

```
def PUT(self, REQUEST, RESPONSE):
    """
    Implement WebDAV/HTTP PUT/FTP put method for this object.
    """
    self.dav__init(REQUEST, RESPONSE)
    self.dav__simpleifhandler(REQUEST, RESPONSE)
    ...
```

Finally your class's edit methods should check to determine whether your object is locked using the `ws_isLocked` method. If someone attempts to change your object when it is locked you should raise the `ResourceLockedError`. For example:

```
from webdav import ResourceLockedError

class MyContentClass(...):
    ...

    def edit(self, ...):
        if self.ws_isLocked():
            raise ResourceLockedError
        ...
```

WebDAV support is not difficult to implement, and as more WebDAV editors become available, it will become more valuable. If you choose to add FTP support to your class you should probably go ahead and support WebDAV too since it is so easy once you've added FTP support.

XML−RPC

XML−RPC is a light−weight Remote Procedure Call protocol that uses XML for encoding and HTTP for transport. Fredrick Lund maintains a Python XML−RPC module.

All objects in Zope support XML−RPC publishing. Generally you will select a published object as the end−point and select one of its methods as the method. For example you can call the `getId` method on a Zope folder at `http://example.com/myfolder` like so:

```
import xmlrpclib
folder=xmlrpclib.Server('http://example.com/myfolder')
ids=folder.getId()
```

You can also do traversal via a dotted method name. For example:

```
import xmlrpclib

# traversal via dotted method name
app=xmlrpclib.Server('http://example.com/app')
id1=app.folderA.folderB.getId()

# walking directly up to the published object
folderB=xmlrpclib.Server('http://example.com/app/folderA/folderB')
id2=folderB.getId()
```

```
        print id1==id2
```

This example shows different routes to the same object publishing call.

XML−RPC supports marshalling of basic Python types for both publishing requests and responses. The upshot of this arrangement is that when you are designing methods for use via XML−RPC you should limit your arguments and return values to simple values such as Python strings, lists, numbers and dictionaries. You should not accept or return Zope objects from methods that will be called via XML−RPC.

XML−RPC does not support keyword arguments. This is a problem if your method expect keyword arguments. This problem is noticeable when calling DTMLMethods and DTMLDocuments with XML−RPC. Normally a DTML object should be called with the request as the first argument, and additional variables as keyword arguments. You can get around this problem by passing a dictionary as the first argument. This will allow your DTML methods and documents to reference your variables with the `var` tag. However, you cannot do the following:

```
        <dtml−var expr="REQUEST['argument']">
```

Although the following will work:

```
        <dtml−var expr="_['argument']">
```

This is because in this case arguments *are* in the DTML namespace, but they are not coming from the web request.

In general it is not a good idea to call DTML from XML−RPC since DTML usually expects to be called from normal HTTP requests.

One thing to be aware of is that Zope returns `false` for published objects which return None since XML−RPC has no concept of null.

Another issue you may run into is that `xmlrpclib` does not yet support HTTP basic authentication. This makes it difficult to call protected web resources. One solution is to patch `xmlrpclib`. Another solution is to accept authentication credentials in the signature of your published method.

Summary

Object publishing is a simple and powerful way to bring objects to the web. Two of Zope's most appealing qualities is how it maps objects to URLs, and you don't need to concern yourself with web plumbing. If you wish, there are quite a few details that you can use to customize how your objects are located and published.

Chapter 3: Zope Products

Introduction

Zope *products* extend Zope with new functionality. Products most often provide new addable objects, but they can also extend Zope with new DTML tags, new ZClass base classes, and other services.

There are two ways to create products in Zope, through the web, and with files in the filesystem. In this chapter we are going to look at building products on the file system. For information on through the web products, and ZClasses see the *Zope Book,* Chapter 12.

In comparison to through the web products, filesystem products require more overhead to build, but offer more power and flexibility, and they can be developed with familiar tools such as Emacs and CVS.

Soon we will make the examples referenced in this chapter available for download as an example product. Until that time, you will see references to files in this chapter that are not available yet. This will be made available soon.

Development Process

This chapter begins with a discussion of how you will develop products. We'll focus on common engineering tasks that you'll encounter as you develop products.

Consider Alternatives

Before you jump into the development of a product you should consider the alternatives. Would your problem be better solved with ZClasses, External Methods, or Python Scripts? Products excel at extending Zope with new addable classes of objects. If this does not figure centrally in your solution, you should look elsewhere. Products, like External Methods allow you to write unrestricted Python code on the filesystem.

Starting with Interfaces

The first step in creating a product is to create one or more interfaces which describe the product. See Chapter 1 for more information on interfaces and how to create them.

Creating interfaces before you build an implementation is a good idea since it helps you see your design and assess how well it fulfills your requirements.

Consider this interface for a multiple choice poll component (see Poll.py):

```
from Interface import Base

class Poll(Base):
    "A multiple choice poll"

    def castVote(self, index):
        "Votes for a choice"

    def getTotalVotes(self):
        "Returns total number of votes cast"
```

```
            def getVotesFor(self, index):
                "Returns number of votes cast for a given response"

            def getResponses(self):
                "Returns the sequence of responses"

            def getQuestion(self):
                "Returns the question
```

How you name your interfaces is entirely up to you. Here we've decided not to use an "I" or any other special indicator in the name of the interface.

Implementing Interfaces

After you have defined an interface for your product, the next step is to create a prototype in Python that implements your interface.

Here is a prototype of a `PollImplemtation` class that implements the interface you just examined (see PollImplementation.py):

```
        from Poll import Poll

        class PollImplementation:
            """
            A multiple choice poll, implements the Poll interface.

            The poll has a question and a sequence of responses. Votes
            are stored in a dictionary which maps response indexes to a
            number of votes.
            """

            __implements__=Poll

            def __init__(self, question, responses):
                self._question = question
                self._responses = responses
                self._votes = {}
                for i in range(len(responses)):
                    self._votes[i] = 0

            def castVote(self, index):
                "Votes for a choice"
                self._votes[index] = self._votes[index] + 1

            def getTotalVotes(self):
                "Returns total number of votes cast"
                total = 0
                for v in self._votes.values():
                    total = total + v
                return total

            def getVotesFor(self, index):
                "Returns number of votes cast for a given response"
                return self._votes[index]

            def getResponses(self):
                "Returns the sequence of responses"
                return tuple(self._responses)
            def getQuestion(self):
                "Returns the question"
                return self._question
```

You can use this class interactively and test it. Here's an example of interactive testing:

```
>>> from PollImplementation import PollImplementation
>>> p=PollImplementation("What's your favorite color?", ["Red", "Green", "Blue", "I forge
>>> p.getQuestion()
"What's your favorite color?"
>>> p.getResponses()
('Red', 'Green', 'Blue', 'I forget')
>>> p.getVotesFor(0)
0
>>> p.castVote(0)
>>> p.getVotesFor(0)
1
>>> p.castVote(2)
>>> p.getTotalVotes()
2
>>> p.castVoteFor(4)
Traceback (innermost last):
File "<stdin>", line 1, in ?
File "poll2.py", line 19, in castVote
self._votes[index] = self._votes[index] + 1
KeyError: 4
```

Interactive testing is one of Python's great features. It lets you experiment with your code in a simple but powerful way.

At this point you can do a fair amount of work, testing and refining your interfaces and classes which implement them. See Chapter 7 for more information on testing.

So far you have learned how to create Python classes that are documented with interfaces, and verified with testing. Next you'll examine the Zope product architecture. Then you'll learn how to fit your well crafted Python classes into the product framework.

Building Product Classes

To turn a component into a product you must fulfill many contracts. For the most part these contracts are not yet defined in terms of interfaces. Instead you must subclass from base classes that implement the contracts. This makes building products confusing, and this is an area that we are actively working on improving.

Base Classes

Consider an example product class definition:

```
from Acquisition import Implicit
from Globals import Persistent
from AccessControl.Role import RoleManager
from OFS.SimpleItem import Item

class PollProduct(Implicit, Persistent, RoleManager, Item):
    """
    Poll product class
    """
    ...
```

The order of the base classes depends on which classes you want to take precedence over others. Most Zope classes do not define similar names, so you usually don't need to worry about what order these classes are used in your product. Let's take a look at each of these base classes:

`Acquisition.Implicit`

This is the normal acquisition base class. See the *API Reference* for the full details on this class. Many Zope services such as object publishing and security use acquisition, so inheriting from this class is required for products. Actually, you can choose to inherit from `Acquisition.Explicit` if you prefer, however, it will prevent folks from dynamically binding Python Scripts and DTML Methods to instances of your class. In general you should subclass from `Acquisition.Implicit` unless you have a good reason not to.

`Globals.Persistent`

This base class makes instances of your product persistent. For more information on persistence and this class see Chapter 4.

In order to make your poll class persistent you'll need to make one change. Since `_votes` is a dictionary this means that it's a mutable non−persistent sub−object. You'll need to let the persistence machinery know when you change it:

```
def castVote(self, index):
    "Votes for a choice"
    self._votes[index] = self._votes[index] + 1
    self._p_changed = 1
```

The last line of this method sets the `_p_changed` attribute to 1. This tells the persistence machinery that this object has changed and should be marked as `dirty`, meaning that its new state should be written to the database at the conclusion of the current transaction. A more detailed explanation is given in the Persistence chapter of this guide.

`OFS.SimpleItem.Item`

This base class provides your product with the basics needed to work with the Zope management interface. By inheriting from `Item` your product class gains a whole host of features: the ability to be cut and pasted, capability with management views, WebDAV support, basic FTP support, undo support, ownership support, and traversal controls. It also gives you some standard methods for management views and error display including `manage_main()`. You also get the `getId()`, `title_or_id()`, `title_and_id()` methods and the `this()` DTML utility method. Finally this class gives your product basic *dtml−tree* tag support. `Item` is really an everything−but−the−kitchen−sink kind of base class.

`Item` requires that your class and instances have some management interface related attributes.

`meta_type`
: This attribute should be a short string which is the name of your product class as it appears in the product add list. For example the poll product class could have a `meta_type` of `Poll`.

`id` *or* *`__name__`*
: All `Item` instances must have an `id` string attribute which uniquely identifies the instance within it's container. As an alternative you may use `__name__` instead of `id`.

`title`

All `Item` instances must have a `title` string attribute. A title may be an empty string if your instance does not have a title.

In order to make your poll class work correctly as an `Item` you'll need to make a few changes. You must add a `meta_type` class attribute, and you may wish to add an `id` parameter to the constructor:

```
class PollProduct(Item, ...):

    meta_type='Poll'
    ...

    def __init__(self, id, question, responses):
        self.id=id
        self._question = question
        self._responses = responses
        self._votes = {}
        for i in range(len(responses)):
            self._votes[i] = 0
```

Finally, you should probably place `Item` last in your list of base classes. The reason for this is that `Item` provides defaults that other classes such as `ObjectManager` and `PropertyManager` override. By placing other base classes before `Item` you allow them to override methods in `Item`.

AccessControl.Role.RoleManager

This class provides your product with the ability to have its security policies controlled through the web. See Chapter 6 for more information on security policies and this class.

OFS.ObjectManager

This base class gives your product the ability to contain other `Item` instances. In other words, it makes your product class like a Zope folder. This base class is optional. See the *API Reference* for more details. This base class gives you facilities for adding Zope objects, importing and exporting Zope objects, WebDAV, and FTP. It also gives you the `objectIds`, `objectValues`, and `objectItems` methods.

`ObjectManager` makes few requirements on classes that subclass it. You can choose to override some of its methods but there is little that you must do.

If you wish to control which types of objects can be contained by instances of your product you can set the `meta_types` class attribute. This attribute should be a tuple of meta_types. This keeps other types of objects from being created in or pasted into instances of your product. The `meta_types` attribute is mostly useful when you are creating specialized container products.

OFS.PropertyManager

This base class provides your product with the ability to have user−managed instance attributes. See the *API Reference* for more details. This base class is optional.

Your class may specify that it has one or more predefined properties, by specifying a `_properties` class attribute. For example:

```
_properties=({'id':'title', 'type': 'string', 'mode': 'w'},
                    {'id':'color', 'type': 'string', 'mode': 'w'},
                    )
```

The `_properties` structure is a sequence of dictionaries, where each dictionary represents a predefined property. Note that if a predefined property is defined in the `_properties` structure, you must provide an attribute with that name in your class or instance that contains the default value of the predefined property.

Each entry in the `_properties` structure must have at least an `id` and a `type` key. The `id` key contains the name of the property, and the `type` key contains a string representing the object's type. The `type` string must be one of the values: `float`, `int`, `long`, `string`, `lines`, `text`, `date`, `tokens`, `selection`, or `multiple section`. For more information on Zope properties see the *Zope Book*.

For `selection` and `multiple selection` properties, you must include an addition item in the property dictionary, `select_variable` which provides the name of a property or method which returns a list of strings from which the selection(s) can be chosen. For example:

```
_properties=({'id' : 'favorite_color',
              'type' : 'selection',
              'select_variable' : 'getColors'
             },
            )
```

Each entry in the `_properties` structure may optionally provide a `mode` key, which specifies the mutability of the property. The `mode` string, if present, must be `w`, `d`, or `wd`.

A `w` present in the mode string indicates that the value of the property may be changed by the user. A `d` indicates that the user can delete the property. An empty mode string indicates that the property and its value may be shown in property listings, but that it is read−only and may not be deleted.

Entries in the `_properties` structure which do not have a `mode` item are assumed to have the mode `wd` (writable and deleteable).

Security Declarations

In addition to inheriting from a number of standard base classes, you must declare security information in order to turn your component into a product. See Chapter 6 for more information on security and instructions for declaring security on your components.

Here's an example of how to declare security on the poll class:

```
from AccessControl import ClassSecurityInfo

class PollProduct(...):
    ...

    security=ClassSecurityInfo()

    security.declareProtected('Use Poll', 'castVote')
    def castVote(self, index):
        ...

    security.declareProtected('View Poll results', 'getTotalVotes')
    def getTotalVotes(self):
        ...
    security.declareProtected('View Poll results', 'getVotesFor')
    def getVotesFor(self, index):
        ...

    security.declarePublic('getResponses')
    def getResponses(self):
        ...

    security.declarePublic('getQuestion')
    def getQuestion(self):
        ...
```

For security declarations to be set up Zope requires that you initialize your product class. Here's how to initialize your poll class:

```
    from Globals import InitializeClass

    class PollProduct(...):
       ...

    InitializeClass(PollProduct)
```

Summary

Congratulations, you've created a product class. Here it is in all its glory (see PollProduct.py):

```
    from Poll import Poll
    from AccessControl import ClassSecurityInfo
    from Globals import InitializeClass
    from Acquisition import Implicit
    from Globals import Persistent
    from AccessControl.Role import RoleManager
    from OFS.SimpleItem import Item

    class PollProduct(Implicit, Persistent, RoleManager, Item):
        """
        Poll product class, implements Poll interface.

        The poll has a question and a sequence of responses. Votes
        are stored in a dictionary which maps response indexes to a
        number of votes.
        """

        __implements__=Poll

        meta_type='Poll'

        security=ClassSecurityInfo()

        def __init__(self, id, question, responses):
            self.id=id
            self._question = question
            self._responses = responses
            self._votes = {}
            for i in range(len(responses)):
                self._votes[i] = 0

        security.declareProtected('Use Poll', 'castVote')
        def castVote(self, index):
            "Votes for a choice"
            self._votes[index] = self._votes[index] + 1
            self._votes = self._votes

        security.declareProtected('View Poll results', 'getTotalVotes')
        def getTotalVotes(self):
            "Returns total number of votes cast"
            total = 0
            for v in self._votes.values():
                total = total + v
            return total

        security.declareProtected('View Poll results', 'getVotesFor')
        def getVotesFor(self, index):
            "Returns number of votes cast for a given response"
            return self._votes[index]

        security.declarePublic('getResponses')
        def getResponses(self):
            "Returns the sequence of responses"
            return tuple(self._responses)

        security.declarePublic('getQuestion')
        def getQuestion(self):
            "Returns the question"
            return self._question

    InitializeClass(Poll)
```

Now it's time to test your product class in Zope. To do this you must register your product class with Zope.

Registering Products

Products are Python packages that live in `lib/python/Products`. Products are loaded into Zope when Zope starts up. This process is called *product initialization*. During product initialization, each product is given a chance to register its capabilities with Zope.

Product Initialization

When Zope starts up it imports each product and calls the product's `initialize` function passing it a registrar object. The `initialize` function uses the registrar to tell Zope about its capabilities. Here is an example `__init__.py` file:

```
    from PollProduct import PollProduct, addForm, addFunction

    def initialize(registrar):
        registrar.registerClass(
            PollProduct,
            constructors = (addForm, addFunction),
            )
```

This function makes one call to the registrar object which registers a class as an addable object. The registrar figures out the name to put in the product add list by looking at the `meta_type` of the class. Zope also deduces a permission based on the class's meta−type, in this case *Add Polls* (Zope automatically pluralizes "Poll" by adding an "s"). The `constructors` argument is a tuple of objects consisting of an add form which is called when a user selects the object from the product add list, and the add method which is the method called by the add form.

Note that you cannot restrict which types of containers can contain instances of your classes. In other words, when you register a class, it will appear in the product add list in folders if the user has the constructor permission.

See the *API Reference* for more information on the `ProductRegistrar` interface.

Factories and Constructors

Factories allow you to create Zope objects that can be added to folders and other object managers. Factories are discussed in Chapter 12 of the *Zope Book*. The basic work a factory does is to put a name into the product add list and associate a permission and an action with that name. If you have the required permission then the name will appear in the product add list, and when you select the name from the product add list, the action method will be called.

Products use Zope factory capabilities to allow instances of product classes to be created with the product add list. In the above example of product initialization you saw how a factory is created by the product registrar. Now let's see how to create the add form and the add list.

The add form is a function that returns an HTML form that allows a users to create an instance of your product class. Typically this form collects that id and title of the instance along with other relevant data. Here's a very simple add form function for the poll class:

```
def addForm():
    return """<html>
    <head><title>Add Poll</title></head>
    <body>
    <form action="addFunction">
    id <input type="type" name="id"><br>
    question <input type="type" name="question"><br>
    responses (one per line)
    <textarea name="responses:lines"></textarea>
    </form>
    </body>
    </html>"""
```

Notice how the action of the form is `addFunction`. Also notice how the lines of the response are marshalled into a sequence. See Chapter 2 for more information about argument marshalling and object publishing.

It's also important to include a HTML `head` tag in the add form. This is necessary so that Zope can set the base URL to make sure that the relative link to the `addFunction` works correctly.

The add function will be passed a `FactoryDispatcher` as its first argument which proxies the location (usually a Folder) where your product was added. The add function may also be passed any form variables which are present in your add form according to normal object publishing rules.

Here's an add function for your poll class:

```
def addFunction(dispatcher, id, question, responses):
    """
    Create a new poll and add it to myself
    """
    p=PollProduct(id, question, responses)
    dispatcher.Destination()._setObject(id, p)
```

The dispatcher has three methods:

`Destination`
    The `ObjectManager` where your product was added.

`DestinationURL`
    The URL of the `ObjectManager` where your product was added.

`manage_main`
    Redirects to a management view of the `ObjectManager` where your product was added.

Notice how it calls the `_setObject()` method of the destination `ObjectManager` class to add the poll to the folder. See the *API Reference* for more information on the `ObjectManager` interface.

The add function should also check the validity of its input. For example the add function should complain if the question or response arguments are not of the correct type.

Testing

Now you're ready to register your product with Zope. You need to add the add form and add method to the poll module. Then you should create a `Poll` directory in your `lib/python/Products` directory and add the `Poll.py`, `PollProduct.py`, and `__init__.py` files. Then restart Zope.

Now login to Zope as a manager and visit the web management interface. You should see a `Poll` product listed inside the *Products* folder in the *Control_Panel*. If Zope had trouble initializing your product you will see a traceback here. Fix your problems, if any and restart Zope. If you are tired of all this restarting, take a look at the *Refresh* facility covered in Chapter 7.

Now go to the root folder. Select *Poll* from the product add list. Notice how you are taken to the add form. Provide an id, a question, and a list of responses and click *Add*. Notice how you get a black screen. This is because your add method does not return anything. Notice also that your poll has a broken icon, and only has the management views. Don't worry about these problems now, you'll find out how to fix these problems in the next section.

Now you should build some DTML Methods and Python Scripts to test your poll instance. Here's a Python Script to figure out voting percentages:

```
## Script (Python) "getPercentFor"
##parameters=index
##
"""
Returns the percentage of the vote given a response index. Note,
this script should be bound a poll by acquisition context.
"""
poll=context
return float(poll.getVotesFor(index)) / poll.getTotalVotes()
```

Here's a DTML Method that displays poll results and allows you to vote:

```
<dtml-var standard_html_header>

<h2>
  <dtml-var getQuestion>
</h2>
```

```
<form> <!-- calls this dtml method -->

<dtml-in getResponses>
  <p>
    <input type="radio" name="index" value="&dtml-sequence-index;">
    <dtml-var sequence-item>
  </p>
</dtml-in>

<input type="submit" value=" Vote ">

</form>

<!-- process form -->

<dtml-if index>
  <dtml-call expr="castVote(index)">
</dtml-if>

<!-- display results -->

<h2>Results</h2>

<p><dtml-var getTotalVotes> votes cast</p>

<dtml-in getResponses>
  <p>
    <dtml-var sequence-item> -
    <dtml-var expr="getPercentFor(_.get('sequence-index'))">%
  </p>
</dtml-in>

<dtml-var standard_html_footer>
```

To use this DTML Method, call it on your poll instance. Notice how this DTML makes calls to both your poll instance and the `getPercentFor` Python script.

At this point there's quite a bit of testing and refinement that you can do. Your main annoyance will be having to restart Zope each time you make a change to your product class (but see Chapter 7 for information on how to avoid all this restarting). If you vastly change your class you may break existing poll instances, and will need to delete them and create new ones. See Chapter 7 for more information on debugging techniques which will come in handy.

Building Management Interfaces

Now that you have a working product let's see how to beef up its user interface and create online management facilities.

Defining Management Views

All Zope products can be managed through the web. Products have a collection of management tabs or *views* which allow managers to control different aspects of the product.

A product's management views are defined in the `manage_options` class attribute. Here's an example:

```
manage_options=(
            {'label' : 'Edit', 'action' : 'editMethod'},
            {'label' : 'View', 'action' : 'viewMethod'},
            )
```

The `manage_options` structure is a tuple that contains dictionaries. Each dictionary defines a management view. The view dictionary can have a number of items.

`label`
: This is the name of the management view

`action`
: This is the URL that is called when the view is chosen. Normally this is the name of a method that displays a management view.

`target`
: An optional target frame to display the action. This item is rarely needed.

`help`
: Optional help information associated with the view. You'll find out more about this option later.

Management views are displayed in the order they are defined. However, only those management views for which the current user has permissions are displayed. This means that different users may see different management views when managing your product.

Normally you will define a couple custom views and reusing some existing views that are defined in your base classes. Here's an example:

```
class PollProduct(Item, ...):
    ...

    manage_options=(
        {'label' : 'Edit', 'action' : 'editMethod'},
        {'label' : 'Options', 'action' : 'optionsMethod'},
        ) + RoleManager.manage_options + Item.manage_options
```

This example would include the standard management view defined by `RoleManager` which is *Security* and those defined by `Item` which are *Undo* and *Ownership*. You should include these standard management views unless you have good reason not to. If your class has a default view method ('index_html') you should also include a *View* view whose action is an empty string. See Chapter 2 for more information on `index_html`.

Note: you should not make the *View* view the first view on your class. The reason is that the first management view is displayed when you click on an object in the Zope management interface. If the *View* view is displayed first, users will be unable to navigate to the other management views since the view tabs will not be visible.

Creating Management Views

The normal way to create management view methods is to use DTML. You can use the `DTMLFile` class to create a DTML Method from a file. For example:

```
from Globals import DTMLFile

class PollProduct(...):
  ...

  editForm=DTMLFile('dtml/edit', globals())
```

```
          ...
```

This creates a DTML Method on your class which is defined in the `dtml/edit.dtml` file. Notice that you do not have to include the `.dtml` file extension. Also, don't worry about the forward slash as a path separator; this convention will work fine on Windows. By convention DTML files are placed in a `dtml` subdirectory of your product. The `globals()` argument to the `DTMLFile` constructor allows it to locate your product directory. If you are running Zope in debug mode then changes to DTML files are reflected right away. In other words you can change the DTML of your product's views without restarting Zope to see the changes.

DTML class methods are callable directly from the web, just like other methods. So now users can see your edit form by calling the `editForm` method on instances of your poll class. Typically DTML methods will make calls back to your instance to gather information to display. Alternatively you may decide to wrap your DTML methods with normal methods. This allows you to calculate information needed by your DTML before you call it. This arrangement also ensures that users always access your DTML through your wrapper. Here's an example:

```
        from Globals import DTMLFile

        class PollProduct(...):
          ...

          _editForm=DTMLFile('dtml/edit', globals())

          def editForm(self, ...):
              ...

              return self._editForm(REQUEST, ...)
```

When creating management views you should include the DTML variables `manage_page_header` and `manage_tabs` at the top, and `manage_page_footer` at the bottom. These variables are acquired by your product and draw a standard management view header, tabs widgets, and footer. The management header also includes CSS information which you can take advantage of if you wish to add CSS style information to your management views. The management CSS information is defined in the `lib/python/App/dtml/manage_page_style.css.dtml` file. Here are the CSS classes defined in this file and conventions for their use.

`form-help`
: Explanatory text related to forms. In the future, users may have the option to hide this text.

`std-text`
: Declarative text unrelated to forms. You should rarely use this class.

`form-title`
: Form titles.

`form-label`
: Form labels for required form elements.

`form-optional`
: Form labels for optional form elements.

`form-element`
: Form elements. Note, because of a Netscape bug, you should not use this class on `textarea` elements.

`form-text`
: Declarative text in forms.

`form-mono`

Fixed width text in forms. You should rarely use this class.

Here's an example management view for your poll class. It allows you to edit the poll question and responses (see 'editPollForm.dtml'):

```
<dtml-var manage_page_header>
<dtml-var manage_tabs>

<p class="form-help">
This form allows you to change the poll's question and
responses. <b>Changing a poll's question and responses
will reset the poll's vote tally.</b>.
</p>

<form action="editPoll">
<table>

  <tr valign="top">
    <th class="form-label">Question</th>
    <td><input type="text" name="question" class="form-element"
    value="&dtml-getQuestion;"></td>
  </tr>

  <tr valign="top">
    <th class="form-label">Responses</th>
    <td><textarea name="responses:lines" cols="50" rows="10">
    <dtml-in getResponses>
    <dtml-var sequence-item html_quote>
    </dtml-in>
    </textarea>
    </td>
  </tr>

  <tr>
    <td></td>
    <td><input type="submit" value="Change" class="form-element"></td>
  </tr>

</table>
</form>

<dtml-var manage_page_header>
```

This DTML method displays an edit form that allows you to change the questions and responses of your poll. Notice how poll properties are HTML quoted either by using `html_quote` in the `dtml-var` tag, or by using the `dtml-var` entity syntax.

Assuming this DTML is stored in a file `editPollForm.dtml` in your product's `dtml` directory, here's how to define this method on your class:

```
class PollProduct(...):
    ...

    security.declareProtected('View management screens', 'editPollForm')
    editPollForm=DTML('dtml/editPollForm', globals())
```

Notice how the edit form is protected by the 'View management screens' permission. This ensures that only managers will be able to call this method.

Notice also that the action of this form is `editPoll`. Since the poll as it stands doesn't include any edit methods you must define one to accept the changes. Here's an `editPoll` method:

```
class PollProduct(...):
    ...

    def __init__(self, id, question, responses):
        self.id=id
        self.editPoll(question, response)

    ...

    security.declareProtected('Change Poll', 'editPoll')
    def editPoll(self, question, responses):
        """
        Changes the question and responses.
        """
        self._question = question
        self._responses = responses
        self._votes = {}
        for i in range(len(responses)):
            self._votes[i] = 0
```

Notice how the `__init__` method has been refactored to use the new `editPoll` method. Also notice how the `editPoll` method is protected by a new permissions, `Change Poll`.

There still is a problem with the `editPoll` method. When you call it from the `editPollForm` through the web nothing is returned. This is a bad management interface. You want this method to return an HTML response when called from the web, but you do not want it to do this when it is called from `__init__`. Here's the solution:

```
class Poll(...):
    ...

    def editPoll(self, question, responses, REQUEST=None):
        """
        Changes the question and responses.
        """
        self._question = question
        self._responses = responses
        self._votes = {}
        for i in range(len(responses)):
            self._votes[i] = 0
        if REQUEST is not None:
            return self.editPollForm(REQUEST,
                manage_tabs_message='Poll question and responses changed.')
```

If this method is called from the web, then Zope will automatically supply the `REQUEST` parameter. (See chapter 2 for more information on object publishing). By testing the `REQUEST` you can find out if your method was called from the web or not. If you were called from the web you return the edit form again.

A management interface convention that you should use is the `manage_tab_message` DTML variable. If you set this variable when calling a management view, it displays a status message at the top of the page. You should use this to provide feedback to users indicating that their actions have been taken when it is not obvious. For example if you don't return a status message from your `editPoll` method, users may be confused and may not realize that their changes have been made.

Sometimes when displaying management views, the wrong tab will be highlighted. This is because `manage_tabs` can't figure out from the URL which view should be highlighted. The solution is to set the `management_view` variable to the label of the view that should be highlighted. Here's an example, using the `editPoll` method:

```
        def editPoll(self, question, responses, REQUEST=None):
            """
            Changes the question and responses.
            """
            self._question = question
            self._responses = responses
            self._votes = {}
            for i in range(len(responses)):
                self._votes[i] = 0
            if REQUEST is not None:
                return self.editPollForm(REQUEST,
                    management_view='Edit',
                    manage_tabs_message='Poll question and responses changed.')
```

Now let's take a look a how to define an icon for your product.

Icons

Zope products are identified in the management interface with icons. An icon should be a 16 by 16 pixel GIF image with a transparent background. Normally icons files are located in a `www` subdirectory of your product package. To associate an icon with a product class, use the `icon` parameter to the `registerClass` method in your product's constructor. For example:

```
        def initialize(registrar):
            registrar.registerClass(
                PollProduct,
                constructors = (addForm, addFunction),
                icon = 'www/poll.gif'
                )
```

Notice how in this example, the icon is identified as being within the product's `www` subdirectory.

See the *API Reference* for more information on the `registerClass` method of the `ProductRegistrar` interface.

Online Help

Zope has an online help system that you can use to provide help for your products. Its main features are context−sensitive help and API help. You should provide both for your product.

Context Sensitive Help

To create context sensitive help, create one help file per management view in your product's `help` directory. You have a choice of formats including: HTML, DTML, structured text, GIF, JPG, and PNG.

Register your help files at product initialization with the `registerHelp()` method on the registrar object:

```
          def initialize(registrar):
              ...
```

```
            registrar.registerHelp()
```

This method will take care of locating your help files and creating help topics for each help file. It can recognize these file extensions: `.html`, `.htm`, `.dtml`, `.txt`, `.stx`, `.gif`, `.jpg`, `.png`.

If you want more control over how your help topics are created you can use the `registerHelpTopic()` method which takes an id and a help topic object as arguments. For example:

```
from mySpecialHelpTopics import MyTopic

def initialize(context):
    ...
    context.registerHelpTopic('myTopic', MyTopic())
```

Your help topic should adhere to the `HelpTopic` interface. See the *API Reference* for more details.

The chief way to bind a help topic to a management screen is to include information about the help topic in the class's manage_options structure. For example:

```
manage_options=(
    {'label':'Edit',
     'action':'editMethod',
     'help':('productId','topicId')},
    )
```

The `help` value should be a tuple with the name of your product's Python package, and the file name (or other id) of your help topic. Given this information, Zope will automatically draw a *Help* button on your management screen and link it to your help topic.

To draw a help button on a management screen that is not a view (such as an add form), use the `HelpButton` method of the `HelpSys` object like so:

```
<dtml-var "HelpSys.HelpButton('productId', 'topicId')">
```

This will draw a help button linked to the specified help topic. If you prefer to draw your own help button you can use the helpURL method instead like so:

```
<dtml-var "HelpSys.helpURL(
  topic='productId',
  product='topicId')">
```

This will give you a URL to the help topic. You can choose to draw whatever sort of button or link you wish.

Other User Interfaces

In addition to providing a through the web management interface your products may also support many other user interfaces. You product might have no web management interfaces, and might be controlled completely through some other network protocol. Zope provides interfaces and support for FTP, WebDAV and XML−RPC. If this isn't enough you can add other protocols.

FTP and WebDAV Interfaces

Both FTP and WebDAV treat Zope objects like files and directories. See Chapter 2 for more information on FTP and WebDAV.

By simply sub−classing from `SimpleItem.Item` and `ObjectManager` if necessary, you gain basic FTP and WebDAV support. Without any work your objects will appear in FTP directory listings and if your class is an `ObjectManager` its contents will be accessible via FTP and WebDAV. See Chapter 2 for more information on implementing FTP and WebDAV support.

XML−RPC and Network Services

XML−RPC is covered in Chapter 2. All your product's methods can be accessible via XML−RPC. However, if your are implementing network services, you should explicitly plan one or more methods for use with XML−RPC.

Since XML−RPC allows marshalling of simple strings, lists, and dictionaries, your XML−RPC methods should only accept and return these types. These methods should never accept or return Zope objects. XML−RPC also does not support `None` so you should use zero or something else in place of `None`.

Another issue to consider when using XML−RPC is security. Many XML−RPC clients still don't support HTTP basic authorization. Depending on which XML−RPC clients you anticipate, you may wish to make your XML−RPC methods public and accept authentication credentials as arguments to your methods.

Content Management Framework Interface

The Content Management Framework is an evolving content management extension for Zope. It provides a number of interfaces and conventions for content objects. If you wish to support the CMF you should consult the CMF user interface guidelines and interface documentation.

Supporting the CMF interfaces is not a large burden if you already support the Zope management interface. You should consider supporting the CMF if your product class handles user manageable content such as documents, images, business forms, etc.

Packaging Products

Zope products are normally packaged as tarballs. You should create your product tarball in such a way as to allow it to be unpacked in the Zope directory. For example, `cd` to the Zope directory and then issue a `tar` comand like so:

```
$ tar cvfz MyProduct−1.0.1.tgz lib/python/Products/MyProduct
```

This will create a gzipped tar archive containing your product. You should include your product name and version number in file name of the archive.

See the Poll−1.0.tgz file for an example of a fully packaged Python product.

Product Information Files

Along with your Python and DTML files you should include some information about your product in its root directory.

`README.txt`
: Provides basic information about your product. Zope will parse this file as structured text and make it available on the *README* view of your product in the control panel.

`VERSION.txt`
: Contains the name and version of your product on a single line. For example, `Mutiple Choice Poll 1.1.0`. Zope will display this information as the `version` property of your product in the control panel.

`LICENSE.txt`
: Contains your product license, or a link to it.

You may also wish to provide additional information. Here are some suggested optional files to include with your product.

`INSTALL.txt`
: Provides special instructions for installing the product and components on which it depends. This file is only optional if your product does not require more than an ungzip/untar into a Zope installation to work.

`TODO.txt`
: This file should make clear where this product release needs work, and what the product author intends to do about it.

`CHANGES.txt` and `HISTORY.txt`
: `CHANGES.txt` should enumerate changes made in particular product versions from the last release of the product. Optionally, a `HISTORY.txt` file can be used for older changes, while `CHANGES.txt` lists only recent changes.

`DEPENDENCIES.txt`
: Lists dependencies including required os platform, required Python version, required Zope version, required Python packages, and required Zope products.

Product Directory Layout

By convention your product will contain a number of sub−directories. Some of these directories have already been discussed in this chapter. Here is a summary of them.

`dtml`
: Contains your DTML files.

`www`
: Contains your icon files.

`help`
: Contains your help files.

`tests`
: Contains your unit tests.

It is not necessary to include these directories if your don't have anything to go in them.

Product Frameworks

Creating Zope products is a complex business. There are a number of frameworks available to help ease the burden of creating products. Different frameworks focus on different aspects of product construction.

ZClass Base Classes

As an alternative to creating full blown products you may choose to create Python base classes which can be used by ZClasses. This allows you to focus on application logic and use ZClasses to take care of management interface issues.

The chief drawback to this approach is that your code will be split between a ZClass and a Python base class. This makes it harder to edit and to visualize.

See the *Zope Book* for more information on ZClasses.

TransWarp and ZPatterns

TransWarp and ZPatterns are two related product framework packages by Phillip Eby and Ty Sarna. You can find out more information on TransWarp from the TransWarp Home Page. More information on ZPatterns can be found at the ZPatterns Home Page

MetaPublisher

MetaPublisher is a content managment system based on Zope technology. It is developed by the Zope solution's provider Beehive. More information can be found on the MetaPublisher Home Page.

Evolving Products

As you develop your product classes you will generally make a series of product releases. While you don't know in advance how your product will change, when it does change there are measures that you can take to minimize problems.

Evolving Classes

Issues can occur when you change your product class because instances of these classes are generally persistent. This means that instances created with an old class will start using a new class. If your class changes drastically this can break existing instances.

The simplest way to handle this situation is to provide class attributes as defaults for newly added attributes. For example if the latest version of your class expects an `improved_spam` instance attribute while earlier versions only sported `spam` attributes, you may wish to define an `improved_spam` class attribute in your new class so your old objects won't break when they run with your new class. You might set `improved_spam` to None in your class, and in methods where you use this attribute you may have to take into account that it may be None. For example:

```
class Sandwich(...):

    improved_spam=None
    ...

    def assembleSandwichMeats(self):
        ...
        # test for old sandwich instances
        if self.improved_spam is None:
            self.updateToNewSpam()
        ...
```

Another solution is to use the standard Python pickling hook `__setstate__`, however, this is in general more error prone and complex.

A third option is to create a method to update old instances. Then you can manually call this method on instances to update to them. Note, this won't work unless the instances function well enough to be accessible via the Zope management screens.

While you are developing a product you won't have to worry too much about these details, since you can always delete old instances that break with new class definitions. However, once you release your product and other people start using it, then you need to start planning for the eventuality of upgrading.

Another nasty problem that can occur is breakage caused by renaming your product classes. You should avoid this since it breaks all existing instances. If you really must change your class name, provide aliases to it using the old name. You may however, change your class's base classes without causing these kinds of problems.

Evolving Interfaces

The basic rule of evolving interfaces is *don't do it*. While you are working privately you can change your interfaces all you wish. But as soon as you make your interfaces public you should freeze them. The reason is that it is not fair to users of your interfaces to changes them after the fact. An interface is contract. It specifies how to use a component and it specifies how to implement types of components. Both users and developers will have problems if your change the interfaces they are using or implementing.

The general solution is to create simple interfaces in the first place, and create new ones when you need to change an existing interface. If your new interfaces are compatible with your existing interfaces you can indicate this by making your new interfaces extend your old ones. If your new interface replaces an old one but does not extend it you should give it a new name such as, `WidgetWithBellsOn`. Your components should continue to support the old interface in addition to the new one for a few releases.

Conclusion

Migrating your components into fully fledged Zope products is a process with a number of steps. There are many details to keep track of. However, if you follow the recipe laid out in this chapter you should have no problems.

As Zope grows and evolves we want to simplify the Zope development model. We hope to remove much of the management interface details from product development. We also want to move to a fuller component framework that makes better use of interfaces.

Nevertheless, Zope products are a powerful framework for building web applications. By creating products you can take advantage of Zope's features including security, scalability, through the web management, and collaboration.

Chapter 4: ZODB Persistent Components

Most Zope components live in the Zope Object DataBase (ZODB). Components that are stored in ZODB are said to be *persistent*. Creating persistent components is, for the most part, a trivial exercise, but ZODB does impose a few rules that persistent components must obey in order to work properly. This chapter describes the persistence model and the interfaces that persistent objects can use to live inside the ZODB.

Persistent Objects

Persistent objects are Python objects that live for a long time. Most objects are created when a program is run and die when the program finishes. Persistent objects are not destroyed when the program ends, they are saved in a database.

A great benefit of persistent objects is their transparency. As a developer, you do not need to think about loading and unloading the state of the object from memory. Zope's persistent machinery handles all of that for you.

This is also a great benefit for application designers; you do not need to create your own kind of "data format" that gets saved to a file and reloaded again when your program stops and starts. Zope's persistence machinery works with *any* kind of Python objects (within the bounds of a few simple rules) and as your types of objects grow, your database simply grows transparently with it.

Persistence Example

Here is a simple example of using ZODB outside of Zope. If all you plan on doing is using persistent objects with Zope, you can skip this section if you wish.

The first thing you need to do to start working with ZODB is to create a "root object". This process involves first opening a "storage", which is the actual backend storage location for your data.

ZODB supports many pluggable storage back−ends, but for the purposes of this article we're going to show you how to use the `FileStorage` back−end storage, which stores your object data in a file. Other storages include storing objects in relational databases, Berkeley databases, and a client to server storage that stores objects on a remote storage server.

To set up a ZODB, you must first install it. ZODB comes with Zope, so the easiest way to install ZODB is to install Zope and use the ZODB that comes with your Zope installation. For those of you who don't want all of Zope, but just ZODB, see the instructions for downloading ZODB from the ZODB web page.

After installing ZODB, you can start to experiment with it right from the Python command line interpreter. For example, try the following python code in your interpeter:

```
from ZODB import FileStorage, DB
storage = FileStorage.FileStorage('mydatabase.fs')
db = DB( storage )
connection = db.open()
root = connection.root()
```

Here, you create storage and use the `mydatabse.fs` file to store the object information. Then, you create a database that uses that storage.

Next, the database needs to be "opened" by calling the `open()` method. This will return a connection object to the database. The connection object then gives you access to the `root` of the database with the `root()` method.

The `root` object is the dictionary that holds all of your persistent objects. For example, you can store a simple list of strings in the root object:

```
root['employees'] = ['Bob', 'Mary', 'Jo']
```

Now, you have changed the persistent database by adding a new object, but this change is so far only temporary. In order to make the change permanent, you must commit the current transaction:

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

Transactions are ways to make a lot of changes in one atomic operation. In a later article, we'll show you how this is a very powerful feature. For now, you can think of committing transactions as "checkpoints" where you save the changes you've made to your objects so far. Later on, we'll show you how to abort those changes, and how to undo them after they are committed.

If you had used a relational database, you would have had to issue a SQL query to save even a simple python list like the above example. You would have also needed some code to convert a SQL query back into the list when you wanted to use it again. You don't have to do any of this work when using ZODB. Using ZODB is almost completely transparent, in fact, ZODB based programs often look suspiciously simple!

Working with simple python types is useful, but the real power of ZODB comes out when you store your own kinds of objects in the database. For example, consider a class that represents a employee:

```
from Persistence import Persistent

class Employee(Persistent):

    def setName(self, name):
        self.name = name
```

Calling `setName` will set a name for the employee. Now, you can put Employee objects in your database:

```
for name in ['Bob', 'Mary', 'Joe']:
    employee = Employee()
    employee.setName(name)
    root['employees'].append(employee)

get_transaction().commit()
```

Don't forget to call `commit()`, so that the changes you have made so far are committed to the database, and a new transaction is begun.

Persistent Rules

There are a few rules that must be followed when your objects are persistent.

- Your objects, and their attributes, must be "pickleable".
- Your object cannot have any attributes that begin with `_p_`.
- Attributes of your object that begin with `_v_` are "volatile" and are not saved to the database (see

next section).
- You must explicitly signal any changes made to mutable attributes or use persistent versions of mutable objects, like `ZODB.PersistentMapping` (see below for more information on `PersistentMapping`.)

In this section, we'll look at each of these special rules one by one.

The first rules says that your objects must be pickleable. This means that they can be serialized into a data format with the "pickle" module. Most python data types (numbers, lists, dictionaries) can be pickled. Code objects (method, functions, classes) and file objects (files, sockets) *cannot* be pickled. Instances can be persistent objects if:

- They subclass `Persistence.Persistent`
- All of their attributes are pickleable

The second rule is that none of your objects attributes can begin with `_p_`. For example, `_p_b_and_j` would be an illegal object attribute. This is because the persistence machinery reserves all of these names for its own purposes.

The third rule is that all object attributes that begin with `_v_` are "volatile" and are not saved to the database. This means that as long as the persistent object is in Zope memory cache, volatile attributes can be used. When the object is deactivated (removed from memory) volatile attributes are thrown away.

Volatile attributes are useful for data that is good to cache for a while but can often be thrown away and easily recreated. File connections, cached calculations, rendered templates, all of these kinds of things are useful applications of volatile attributes. You must exercise care when using volatile attributes. Since you have little control over when your objects are moved in and out of memory, you never know when your volatile attributes may disappear.

The fourth rule is that you must signal changes to mutable types. This is because persistent objects can't detect when mutable types change, and therefore, doesn't know whether or not to save the persistent object or not.

For example, say you had a list of names as an attribute of your object called `departments` that you changed in a method called `addDepartment`:

```
class DepartmentManager(Persistent):

    def __init__(self):
        self.departments = []

    def addDepartment(self, department):
        self.departments.append(department)
```

When you call the `addDepartment` method you change a mutable type, `departments` but your persistent object will not save that change.

There are two solutions to this problem. First, you can assign a special flag, `_p_changed`:

```
def addDepartment(self, department):
    self.department.append(department)
    self._p_changed = 1
```

Remember, `_p_` attributes do something special to the persistence machinery and are reserved names. Assigning 1 to `_p_changed` tells the persistence machinery that you changed the object, and that it should be saved.

Another technique is to use the mutable attribute as though it were immutable. In other words, after you make changes to a mutable object, reassign it:

```
def addDepartment(self, department):
    departments = self.departments
    departments.append(department)
    self.department = departments
```

Here, the `self.departments` attribute was re−assigned at the end of the function to the "working copy" object `departments`. This technique is cleaner because it doesn't have any explicit `_p_changed` settings in it, but this implicit triggering of the persistence machinery should always be understood, otherwise use the explicit syntax.

A final option is to use persistence−aware mutable attributes such as `PersistentMapping`, and `IOBTree`. `PersistentMapping` is a mapping class that notifies ZODB when you change the mapping. You can use instances of `PersistentMapping` in place of standard Python dictionaries and not worry about signaling change by reassigning the attribute or using `_p_changed`. Zope's Btree classes are also persistent−aware mutable containers. This solution can be cleaner than using mutable objects immutably, or signaling change manually assuming that there is a persistence−aware class available that meets your needs.

Transactions and Persistent Objects

When changes are saved to ZODB, they are saved in a *transaction*. This means that either all changes are saved, or none are saved. The reason for this is data consistency. Imagine the following scenario:

1. A user makes a credit card purchase at the sandwich.com website.
2. The bank debits their account.
3. An electronic payment is made to sandwich.com.

Now imagine that an error happens during the last step of this process, sending the payment to sandwich.com. Without transactions, this means that the account was debited, but the payment never went to sandwich.com! Obviously this is a bad situation. A better solution is to make all changes in a transaction:

1. A user makes a credit card purchase at the sandwich.com website.
2. The transaction begins
3. The bank debits their account.
4. An electronic payment is made to sandwich.com.
5. The transaction commits

Now, if an error is raised anywhere between steps 2 and 5, *all* changes made are thrown away, so if the payment fails to go to sandwich.com, the account won't be debited, and if debiting the account raises an error, the payment won't be made to sandwich.com, so your data is always consistent.

When using your persistent objects with Zope, Zope will automatically *begin* a transaction when a web request is made, and *commit* the transaction when the request is finished. If an error occurs at any time during that request, then the transaction is *aborted*, meaning all the changes made are thrown away.

If you want to *intentionally* abort a transaction in the middle of a request, then just raise an error at any time. For example, this snippet of Python will raise an error and cause the transaction to abort:

```
raise SandwichError('Not enough peanut butter.')
```

A more likely scenario is that your code will raise an exception when a problem arises. The great thing about transactions is that you don't have to include cleanup code to catch exceptions and undo everything you've done up to that point. Since the transaction is aborted the changes made in the transaction will not be saved.

Because Zope does transaction management for you, most of the time you do not need to explicitly begin, commit or abort your own transactions. For more information on doing transaction management manually, see the links at the end of this chapter that lead to more detailed tutorials of doing your own ZODB programming.

Subtransactions

Zope waits until the transaction is committed to save all the changes to your objects. This means that the changes are saved in memory. If you try to change more objects than you have memory in your computer, your computer will begin to swap and thrash, and maybe even run you out of memory completely. This is bad. The easiest solution to this problem is to not change huge quantities of data in one transaction.

If you need to spread a transaction out of lots of data, however, you can use subtransactions. Subtransactions allow you to manage Zope's memory usage yourself, so as to avoid swapping during large transactions.

Subtransactions allow you to make huge transactions. Rather than being limited by available memory, you are limited by available disk space. Each subtransaction commit writes the current changes out to disk and frees memory to make room for more changes.

To commit a subtransaction, you first need to get a hold of a transaction object. Zope adds a function to get the transaction objects in your global namespace, `get_transaction`, and then call `commit(1)` on the transaction:

```
get_transaction().commit(1)
```

You must balance speed, memory, and temporary storage concerns when deciding how frequently to commit subtransactions. The more subtransactions, the less memory used, the slower the operation, and the more temporary space used. Here's and example of how you might use subtransactions in your Zope code:

```
tasks_per_subtransaction = 10
i = 0
for task in tasks:
    process(task)
    i = i + 1
    if i % tasks_per_subtransaction == 0:
        get_transaction().commit(1)
```

This example shows how to commit a subtransaction at regular intervals while processing a number of tasks.

Threads and Conflict Errors

Zope is a multi−threaded server. This means that many different clients may be executing your Python code in different threads. For most cases, this is not an issue and you don't need to worry about it, but there are a few cases you should look out for.

The first case involves threads making lots of changes to objects and writing to the database. The way ZODB and threading works is that each thread that uses the database gets its own *connection* to the database. Each connection gets its own *copy* of your object. All of the threads can read and change any of the objects. ZODB keeps all of these objects synchronized between the threads. The upshot is that you don't have to do any locking or thread synchronization yourself. Your code can act as through it is single threaded.

However, synchronization problems can occur when objects are changed by two different threads at the same time.

Imagine that thread 1 gets its own copy of object A, as does thread 2. If thread 1 changes its copy of A, then thread 2 will not see those changes until thread 1 commits them. In cases where lots of objects are changing, this can cause thread 1 and 2 to try and commit changes to object 1 at the same time.

When this happens, ZODB lets one transaction do the commit (it "wins") and raises a `ConflictError` in the other thread (which "looses"). The looser can elect to try again, but this may raise yet another `ConflictError` if many threads are trying to change object A. Zope does all of its own transaction management and will retry a losing transaction three times before giving up and raising the `ConflictError` all the way up to the user.

Resolving Conflicts

If a conflict happens, you have two choices. The first choice is that you live with the error and you try again. Statistically, conflicts are going to happen, but only in situations where objects are "hot−spots". Most problems like this can be "designed away"; if you can redesign your application so that the changes get spread around to many different objects then you can usually get rid of the hot spot.

Your second choice is to try and *resolve* the conflict. In many situations, this can be done. For example, consider the following persistent object:

```
class Counter(Persistent):

    self.count = 0

    def hit(self):
        self.count = self.count + 1
```

This is a simple counter. If you hit this counter with a lot of requests though, it will cause conflict errors as different threads try to change the count attribute simultaneously.

But resolving the conflict between conflicting threads in this case is easy. Both threads want to increment the self.count attribute by a value, so the resolution is to increment the attribute by the sum of the two values and make both commits happy; no `ConflictError` is raised.

To resolve a conflict, a class should define an `_p_resolveConflict` method. This method takes three arguments.

`oldState`
: The state of the object that the changes made by the current transaction were based on. The method is permitted to modify this value.

`savedState`
: The state of the object that is currently stored in the database. This state was written after `oldState` and reflects changes made by a transaction that committed before the current transaction. The method is permitted to modify this value.

`newState`
: The state after changes made by the current transaction. The method is *not* permitted to modify this value. This method should compute a new state by merging changes reflected in `savedState` and `newState`, relative to `oldState`.

The method should return the state of the object after resolving the differences.

Here is an example of a `_p_resolveConflict` in the `Counter` class:

```
class Counter(Persistent):

    self.count = 0

    def hit(self):
        self.count = self.count + 1

    def _p_resolveConflict(self, oldState, savedState, newState):

        # Figure out how each state is different:
        savedDiff= savedState['count'] − oldState['count']
        newDiff= newState['count']− oldState['count']

        # Apply both sets of changes to old state:
        oldState['count'] = oldState['count'] + savedDiff + newDiff

        return oldState
```

In the above example, `_p_resolveConflict` resolves the difference between the two conflicting transactions.

Threadsafety of Non−Persistent Objects

ZODB takes care of threadsafety for persistent objects. However, you must handle threadsafey yourself for non−persistent objects which are shared between threads.

Mutable Default Arguments

One tricky type of non−persistent, shared objects are mutable default arguments to functions, and methods. Default arguments are useful because they are cached for speed, and do not need to be recreated every time the method is called. But if these cached default arguments are mutable, one thread may change (mutate) the object when another thread is using it, and that can be bad. So, code like:

```
def foo(bar=[]):
    bar.append('something')
```

Could get in trouble if two threads execute this code because lists are mutable. There are two solutions to this problem:

- Don't use mutable default arguments. (Good)

- If you use them, you cannot change them. If you want to change them, you will need to implement your own locking. (Bad)

We recommend the first solution because mutable default arguments are confusing, generally a bad idea in the first place.

Shared Module Data

Objects stored in modules but not in the ZODB are not persistent and not−thread safe. In general it's not a good idea to store data (as opposed to functions, and class definitions) in modules when using ZODB.

If you decide to use module data which can change you'll need to protect it with a lock to ensure that only one thread at a time can make changes.

For example:

```
from threading import Lock
queue=[]
l=Lock()

def put(obj):
    l.acquire()
    try:
        queue.append(obj)
    finally:
        l.release()

def get():
    l.acquire()
    try:
        return queue.pop()
    finally:
        l.release()
```

Note, in most cases where you are tempted to use shared module data, you can likely achieve the same result with a single persistent object. For example, the above queue could be replaced with a single instance of this class:

```
class Queue(Persistent):

    def __init__(self):
        self.list=[]

    def put(self, obj):
        self.list=self.list + [obj]

    def get(self):
        obj=self.list[−1]
        self.list=self.list[0:−1]
        return obj
```

Notice how this class uses the mutable object `self.list` immutably.

Shared External Resources

A final category of data for which you'll need to handle thread−safety is external resources such as files in the filesystem, and other processes. In practice, these concerns rarely come up.

Other ZODB Resources

This chapter has only covered the most important features of ZODB from a Zope developer's perspective. Check out some of these sources for more in depth information:

- Andrew Kuchling's ZODB pages include lots of information included a programmer's guide and links to ZODB mailing lists.
- ZODB Wiki has information about current ZODB projects.
- ZODB UML Model has the nitty gritty details on ZODB.
- Paper Introduction to the Zope Object Database by Jim Fulton, presented at the 8th Python Conference.

Summary

The ZODB is a complex and powerful system. However using persistent objects is almost completely painless. Seldom do you need to concern yourself with thread safety, transactions, conflicts, memory management, and database replication. ZODB takes care of these things for you. By following a few simple rules you can create persistent objects that just work.

Chapter 5: Acquisition

Acquisition is a mechanism that allows objects to obtain attributes from their environment. It is similar to inheritance, except that, rather than searching an inheritance hierarchy to obtain attributes, a containment hierarchy is traversed.

Introductory Example

Zope implements acquisition with Extension Class mix−in classes. To use acquisition your classes must inherit from an acquisition base class. For example:

```
import ExtensionClass, Acquisition

class C(ExtensionClass.Base):
  color='red'

class A(Acquisition.Implicit):

  def report(self):
    print self.color

a=A()
c=C()
c.a=A()

c.a.report() # prints 'red'

d=C()
d.color='green'
d.a=a

d.a.report() # prints 'green'

a.report() # raises an attribute error
```

The class `A` inherits acquisition behavior from `Acquisition.Implicit`. The object, `a`, "has" the color of objects `c` and `d` when it is accessed through them, but it has no color by itself. The object `a` obtains attributes from its environment, where its environment is defined by the access path used to reach `a`.

Acquisition Wrappers

When an object that supports acquisition is accessed through an extension class instance, a special object, called an acquisition wrapper, is returned. In the example above, the expression `c.a` returns an acquisition wrapper that contains references to both `c` and `a`. It is this wrapper that performs attribute lookup in `c` when an attribute cannot be found in `a`.

Acquisition wrappers provide access to the wrapped objects through the attributes `aq_parent`, `aq_self`, `aq_base`. In the example above, the expressions:

```
'c.a.aq_parent is c'
```

and:

```
'c.a.aq_self is a'
```

both evaluate to true, but the expression:

```
 'c.a is a'
```

evaluates to false, because the expression `c.a` evaluates to an acquisition wrapper around `c` and `a`, not `a` itself.

The attribute `aq_base` is similar to `aq_self`. Wrappers may be nested and `aq_self` may be a wrapped object. The `aq_base` attribute is the underlying object with all wrappers removed.

You can manually wrap object using the `__of__` method. For example:

```
class A(Acquisition.Implicit):
    pass

a=A()
a.color='red'
b=A()
a.b=b

print b.__of__(a).color # prints red
```

The expression `b.__of__(a)` wraps `b` in an acquisition wrapper just like `a.b` does.

Explicit and Implicit Acquisition

Two styles of acquisition are supported: implicit and explicit acquisition.

Implicit acquisition

Implicit acquisition is so named because it searches for attributes from the environment automatically whenever an attribute cannot be obtained directly from an object or through inheritance.

An attribute can be implicitly acquired if its name does not begin with an underscore.

To support implicit acquisition, your class should inherit from the mix−in class `Acquisition.Implicit`.

Explicit Acquisition

When explicit acquisition is used, attributes are not automatically obtained from the environment. Instead, the method `aq_acquire` must be used. For example:

```
  print c.a.aq_acquire('color')
```

To support explicit acquisition, your class should inherit from the mix−in class `Acquisition.Explicit`.

Controlling Acquisition

A class (or instance) can provide attribute by attribute control over acquisition. Your should subclass from `Acquisition.Explicit`, and set all attributes that should be acquired to the special value `Acquisition.Acquired`. Setting an attribute to this value also allows inherited attributes to be overridden with acquired ones. For example:

```
class C(Acquisition.Explicit):
   id=1
   secret=2
   color=Acquisition.Acquired
   __roles__=Acquisition.Acquired
```

The *only* attributes that are automatically acquired from containing objects are `color`, and `__roles__`. Note that the `__roles__` attribute is acquired even though its name begins with an underscore. In fact, the special `Acquisition.Acquired` value can be used in `Acquisition.Implicit` objects to implicitly acquire selected objects that smell like private objects.

Sometimes, you want to dynamically make an implicitly acquiring object acquire explicitly. You can do this by getting the object's `aq_explicit` attribute. This attribute provides the object with an explicit wrapper that places the original implicit wrapper.

Filtered Acquisition

The acquisition method, `aq_acquire`, accepts two optional arguments. The first of the additional arguments is a "filtering" function that is used when considering whether to acquire an object. The second of the additional arguments is an object that is passed as extra data when calling the filtering function and which defaults to `None`. The filter function is called with five arguments:

- The object that the `aq_acquire` method was called on,
- The object where an object was found,
- The name of the object, as passed to `aq_acquire`,
- The object found, and
- The extra data passed to `aq_acquire`.

If the filter returns a true object that the object found is returned, otherwise, the acquisition search continues.

For example, in:

```
from Acquisition import Explicit

class HandyForTesting:
    def __init__(self, name):
        self.name=name
    def __str__(self):
        return "%s(%s)" % (self.name, self.__class__.__name__)
    __repr__=__str__

class E(Explicit, HandyForTesting): pass

class Nice(HandyForTesting):
    isNice=1
    def __str__(self):
        return HandyForTesting.__str__(self)+' and I am nice!'
    __repr__=__str__

a=E('a')
a.b=E('b')
a.b.c=E('c')
a.p=Nice('spam')
a.b.p=E('p')

def find_nice(self, ancestor, name, object, extra):
    return hasattr(object,'isNice') and object.isNice

print a.b.c.aq_acquire('p', find_nice)
```

The filtered acquisition in the last line skips over the first attribute it finds with the name `p`, because the attribute doesn't satisfy the condition given in the filter. The output of the last line is:

```
    spam(Nice) and I am nice!
```

Filtered acquisition is rarely used in Zope.

Acquiring from Context

Normally acquisition allows objects to acquire data from their containers. However an object can acquire from objects that aren't its containers.

Most of the example's we've seen so far show establishing of an acquisition *context* using `getattr` symanitics. For example, `a.b` is a reference to `b` in the context of `a`.

You can also manuallyset acquisition context using the __`of`__ method. For example:

```
    from Acquisition import Implicit
    class C(Implicit): pass
    a=C()
    b=C()
    a.color="red"
    print b.__of__(a).color # prints red
```

In this case, `a` does not contain `b`, but it is put in 'b''s context using the __`of`__ method.

Here's another subtler example that shows how you can construct an acquisition context that includes non−container objects:

```
    from Acquisition import Implicit

    class C(Implicit):
        def __init__(self, name):
            self.name=name

    a=C("a")
    a.b=C("b")
    a.b.color="red"
    a.x=C("x")

    print a.b.x.color # prints red
```

Even though `b` does not contain `x`, `x` can acquire the `color` attribute from `b`. This works because in this case, `x` is accessed in the context of `b` even though it is not contained by `b`.

Here acquisition context is defined by the objects used to access another object.

Containment Before Context

If in the example above suppose both `a` and `b` have an `color` attribute:

```
a=C("a")
a.color="green"
a.b=C("b")
a.b.color="red"
a.x=C("x")

print a.b.x.color # prints green
```

Why does `a.b.x.color` acquire `color` from `a` and not from `b`? The answer is that an object acquires from its containers before non−containers in its context.

To see why consider this example in terms of expressions using the `__of__` method:

```
a.x -> x.__of__(a)

a.b -> b.__of__(a)

a.b.x -> x.__of__(a).__of__(b.__of__(a))
```

Keep in mind that attribute lookup in a wrapper is done by trying to look up the attribute in the wrapped object first and then in the parent object. So in the expressions above proceeds from left to right.

The upshot of these rules is that attributes are looked up by containment before context.

This rule holds true also for more complex examples. For example, `a.b.c.d.e.f.g.attribute` would search for `attribute` in `g` and all its containers first. (Containers are searched in order from the innermost parent to the outermost container.) If the attribute is not found in g or any of its containers, then the search moves to `f` and all its containers, and so on.

Additional Attributes and Methods

You can use the special method `aq_inner` to access an object wrapped only by containment. So in the example above:

```
a.b.x.aq_inner
```

is equivalent to:

```
 a.x
```

You can find out the acquisition context of an object using the `aq_chain` method like so:

```
a.b.x.aq_chain # returns [x, b, a]
```

You can find out if an object is in the acquisition context of another object using the `aq_inContextOf` method. For example:

```
a.b.x.aq_inContextOf(a.b) # returns 1
```

You can also pass an additional argument to `aq_inContextOf` to indicate whether to only check containment rather than the full acquisition context. For example:

```
a.b.x.aq_inContextOf(a.b, 1) # returns 0
```

Note: as of this writing the `aq_inContextOf` examples don't work. According to Jim, this is because `aq_inContextOf` works by comparing object pointer addresses, which (because they are actually different wrapper objects) doesn't give you the expected results. He acknowledges that this behavior is controversial, and says that there is a collector entry to change it so that you would get the answer you expect in the above. (We just need to get to it).

Acquisition Module Functions

In addition to using acquisition attributes and methods directly on objects you can use similar functions defined in the `Acquisition` module. These functions have the advantage that you don't need to check to make sure that the object has the method or attribute before calling it.

`aq_acquire(object, name [, filter, extra, explicit, default, containment])`

Acquires an object with the given name.

This function can be used to explictly acquire when using explicit acquisition and to acquire names that wouldn't normally be acquired.

The function accepts a number of optional arguments:

`filter`

A callable filter object that is used to decide if an object should be acquired.

The filter is called with five arguments:

◊ The object that the aq_acquire method was called on,
◊ The object where an object was found,
◊ The name of the object, as passed to aq_acquire,
◊ The object found, and
◊ The extra argument passed to aq_acquire.

If the filter returns a true object that the object found is returned, otherwise, the acquisition search continues.

`extra`

extra data to be passed as the last argument to the filter.

`explicit`

A flag (boolean value) indicating whether explicit acquisition should be used. The default value is true. If the flag is true, then acquisition will proceed regardless of whether wrappers encountered in the search of the acquisition hierarchy are explicit or implicit wrappers. If the flag is false, then parents of explicit wrappers are not searched.

This argument is useful if you want to apply a filter without overriding explicit wrappers.

`default`

A default value to return if no value can be acquired.

`containment`

A flag indicating whether the search should be limited to the containment hierarchy.

In addition, arguments can be provided as keywords.

`aq_base(object)`
: Return the object with all wrapping removed.

`aq_chain(object [, containment])`
: Return a list containing the object and it's acquisition parents. The optional argument, `containment`, controls whether the containment or access hierarchy is used.

`aq_get(object, name [, default, containment])`
: Acquire an attribute, name. A default value can be provided, as can a flag that limits search to the containment hierarchy.

`aq_inner(object)`
: Return the object with all but the innermost layer of wrapping removed.

`aq_parent(object)`
: Return the acquisition parent of the object or `None` if the object is unwrapped.

`aq_self(object)`
: Return the object with one layer of wrapping removed, unless the object is unwrapped, in which case the object is returned.

In most cases it is more convenient to use these module functions instead of the acquisition attributes and methods directly.

Acquisition and Methods

Python methods of objects that support acquisition can use acquired attributes. When a Python method is called on an object that is wrapped by an acquisition wrapper, the wrapper is passed to the method as the first argument. This rule also applies to user−defined method types and to C methods defined in pure mix−in classes.

Unfortunately, C methods defined in extension base classes that define their own data structures, cannot use aquired attributes at this time. This is because wrapper objects do not conform to the data structures expected by these methods. In practice, you will seldom find this a problem.

Conclusion

Acquisition provides a powerful way to dynamically share information between objects. Zope using acquisition for a number of its key features including security, object publishing, and DTML variable lookup. Acquisition also provides an elegant solution to the problem of circular references for many classes of problems. While acquisition is powerful, you should take care when using acquisition in your applications. The details can get complex, especially with the differences between acquiring from context and acquiring from containment.

Chapter 6: Security

Introduction

A typical web application needs to be securely managed. Different types of users need different kinds of access to the components that make up an application. To this end, Zope includes a comprehensive set of security features. This chapter's goal is to shed light on Zope security in the context of Zope Product development. For a more fundamental overview of Zope security, you may wish to refer to the *Zope Book*, Chapter 6, Users and Security . Before diving into this chapter, you should have a basic understanding of how to build Zope Products as well as an understanding of how the Zope object publisher works. These topics are covered in Chapter 2 and Chapter 3, respectively.

Security Architecture

The Zope security architecture is built around a *security policy*, which you can think of as the "access control philosophy" of Zope. This policy arbitrates the decisions Zope makes about whether to allow or deny access to any particular object defined within the system.

The Zope security policy is consulted when an object is accessed by the publishing machinery and when *restricted code* is run. The Zope security policy is not consulted when *unrestricted code* is run.

How The Security Policy Relates to Zope's Publishing Machinery

When access to Zope is performed via HTTP, WebDAV, or FTP, Zope's publishing machinery consults the security policy in order to determine whether to allow or deny access to a visitor for a particular object. For example, when a user visits the root `index_html` object of your site via HTTP, the security policy is consulted by `ZPublisher` to determine whether the user has permission to view the `index_html` object itself. For more information on this topic, see Chapter 3, "Object Publishing".

How The Security Policy Relates to Restricted Code

Restricted code is generally any sort of logic that may be edited remotely (through the Web, FTP, via WebDAV or by other means). DTML Methods, SQLMethods, Python Scripts and Perl Scripts are examples of restricted code.

When restricted code runs, any access to objects integrated with Zope security is arbitrated by the security policy. For example if you write a bit of restricted code with a line that attempts to manipulate an object you don't have sufficient permission to use, the security policy will deny you access to the object. This generally is accomplished by raising an `Unauthorized` exception, which is a Python string exception caught by a User Folder which signifies that Zope should attempt to get user credentials before obeying the request. The particular code used to attempt to obtain the credentials is determined by the User Folder "closest" (folder−wise) to the object being accessed.

`Unauthorized` Exceptions and Through−The−Web Code

The security policy infrastructure will raise an `Unauthorized` exception automatically when access to an object is denied. When an `Unauthorized` exception is raised within Zope, it is handled in a sane way by Zope, generally by having the User Folder prompt the user for login information. Using this functionality, it's possible to protect Zope objects through access control, only prompting the user for authentication when it is necessary to perform an action which requires privilege.

An example of this behavior can be witnessed within the Zope Management interface itself. The management interface prompts you to log in when visiting, for example, the `/manage` method of any Zope object. This is due to the fact that an anonymous user does not generally possess the proper credntials to use the management interface. If you're using Zope in the default configuration with the default User Folder, it prompts you to provide login information via an HTTP basic authentication dialog.

How The Security Policy Relates To Unrestricted Code

There are also types of *unrestricted code* in Zope, where the logic is not constrained by the security policy. Examples of unrestricted code are the methods of Python classes that implement the objects in Python file−based add−on components. Another example of unrestricted code can be found in External Method objects, which are defined in files on the filesystem. These sorts of code are allowed to run "unrestricted" because access to the file system is required to define such logic. Zope assumes that code defined on the filesystem is "trusted", while code defined "through the web" is not. All filesystem−based code in Zope is unrestricted code.

We'll see later that while the security policy does not constrain what your unrestricted code does, it can and should be used to control the ability to *call* your unrestricted code from within a restricted−code environment.

Details Of The Default Zope Security Policy

In short, the default Zope security policy ensures the following:

- access to an object which does not have any associated security information is always denied.
- if an object is associated with a permission, access is granted or denied based on the user's roles. If a user has a role which has been granted the permission in question, access is granted. If the user does not possess a role that has been granted the permission in question, access is denied.
- if the object has a security assertion declaring it *public* , then access will be granted.
- if the object has a security assertion declaring it *private*, then access will be denied.
- accesses to objects that have names beginning with the underscore character _ are always denied.

As we delve further into Zope security within this chapter, we'll see exactly what it means to associate security information with an object.

Overview Of Using Zope Security Within Your Product

Of course, now that we know what the Zope security policy is, we need to know how our Product can make use of it. Zope developers leverage the Zope security policy primarily by making security declarations related to methods and objects within their Products. Using security assertions, developers may deny or allow all types of access to a particular object or method unilaterally, or they may protect access to Zope objects more granularly by using permissions to grant or deny access based on the roles of the requesting user to the same objects or methods.

For a more fundamental overview of Zope users, roles, and permissions, see the section titled "Authorization and Managing Security" in the Security Chapter of the *Zope Book*.

Security Declarations In Zope Products

Zope security declarations allow developers to make security assertions about a Product−defined object and its methods. Security declarations come in three basic forms. These are:

public
: allow anybody to access access the protected object or method

private
: deny anyone access to the protected object or method

protected
: protect access to the object or method via a permission

We'll see how to actually "spell" these security assertions a little later in this chapter. In the meantime, just know that security declarations are fundamental to Zope Product security, and they can be used to protect access to an object by associating it with a permission. We will refer to security declarations as "declarations" and "assertions" interchangeably within this chapter.

Permissions In Zope Products

A permission is the smallest unit of access to an object in Zope, roughly equivalent to the atomic permissions on files seen in Windows NT or UNIX: R (Read), W(Write), X(Execute), etc. However, unlike these types of mnemonic permissions shared by all sorts of different file types in an operating system product, in Zope, a permission usually describes a fine−grained logical operation which takes place upon an object, such as "View Management Screens" or "Add Properties".

Zope administrators associate these permissions with *roles,* which they grant to Zope users. Thus, declaring a protection assertion on a method of "View management screens" ensures that only users who possess roles which have been granted the "View management screens" permission are able to perform the action that the method defines.

It is important to note that Zope's security architecture dictates that roles and users remain the domain of administrators, while permissions remain the domain of developers. Developers of Products should not attempt to define roles or users, although they may (and usually must) define permissions. Most importantly, a Zope administrator who makes use of your product should have the "last word" as regards which roles are granted which permissions, allowing her to protect her site in a manner that fits her business goals.

Permission names are strings, and these strings are currently arbitrary. There is no permission hierarchy, or list of "approved permissions". Developers are encouraged to reuse Zope core permissions (e.g. "View", "Access contents information") when appropriate, or they may create their own as the need arises. It is generally wise to reuse existing Zope permission names unless you specifically need to define your own. For a list of existing Zope core permissions, see Appendix A, "Zope Core Permissions".

Permissions are often tied to method declarations in Zope. Any number of method declarations may share the same permission. It's useful to declare the same permission on a set of methods which can logically be grouped together. For example, two methods which return management forms for the object can be provided with the same permission, "View management screens". Likewise, two entirely different objects can share a permission name to denote that the operation that's being protected is fundamentally similar. For instance, most Product−defined objects reuse the Zope "View" permission, because most Zope objects need to be viewed in a web browser. If you create an addable Zope class named `MyObject`, it doesn't make much sense to create a permission "View MyObject", because the generic "View" permission may be reused for this action.

There is an exception to the "developers should not try to define roles" rule inasmuch as Zope allows developers to assign "default roles" to a permission. This is primarily for the convenience of the Zope administrator, as default roles for a permission cause the Zope security machinery to provide a permission to a role *by default* when instances of a Product class are encountered during security operations. For example, if your Product defines a permission "Add Poll Objects", this permission may be associated with a set of default roles, perhaps "Manager". Default roles in Products should not be used against roles other than "Manager", "Anonymous", "Owner", and "Authenticated" (the four default Zope roles), as other roles are not guaranteed to exist in every Zope installation.

Using security assertions in Zope is roughly analogous to assigning permission bit settings and ownership information to files in a UNIX or Windows filesystem. Protecting objects via permissions allows developers and administrators to secure Zope objects independently of statements made in application code.

Implementing Security In Python Products

Security Assertions

You may make several kinds of security assertions at the Python level. You do this to declare accessibility of methods and subobjects of your classes. Three of the most common assertions that you'll want to make on your objects are:

- this object is ***public*** (always accessible)
- this object is ***private*** (not accessible by restricted code or by URL traversal)
- this object is ***protected*** by a specific permission

There are a few other kinds of security assertions that are much less frequently used but may be needed in some cases:

- asserting that access to subobjects that do not have explicit security information should be allowed rather than denied.
- asserting what sort of protection should be used when determining access to an *object itself* rather than a particular method of the object

It is important to understand that security assertions made in your Product code *do not* limit the ability of the code that the assertion protects. Assertions only protect *access to this code*. The code which constitutes the body of a protected, private, or public method of a class defined in a Zope disk−based Product runs completely unrestricted, and is not subject to security constraints of any kind within Zope. An exception to this rule occurs when disk−based−Product code calls a "through the web" method such as a Python Script or a DTML Method. In this case, the security constraints imposed by these objects respective to the current request are obeyed.

When Should I Use Security Assertions?

If you are building an object that will be used from DTML or other restricted code, or that will be accessible directly through the web (or other remote protocols such as FTP or WebDAV) then you need to define security information for your object.

Making Security Assertions

As a Python developer, you make security assertions in your Python classes using `SecurityInfo` objects. A `SecurityInfo` object provides the interface for making security assertions about an object in Zope.

The convention of placing security declarations inside Python code may at first seem a little strange if you're used to "plain old Python" which has no notion at all of security declarations. But because Zope provides the ability to make these security assertions at such a low level, the feature is ubiquitous throughout Zope, making it easy to make these declarations once in your code, usable site−wide without much effort.

Class Security Assertions

The most common kind of `SecurityInfo` you will use as a component developer is the `ClassSecurityInfo` object. You use `ClassSecurityInfo` objects to make security assertions about methods on your classes.

Classes that need security assertions are any classes that define methods that can be called "through the web". This means any methods that can be called directly with URL traversal, from DTML Methods, or from Python−based Script objects.

Declaring Class Security

When writing the classes in your product, you create a `ClassSecurityInfo` instance *within each class that needs to play with the security model*. You then use the `ClassSecurityInfo` object to make assertions about your class, its subobjects and its methods.

The `ClassSecurityInfo` class is defined in the `AccessControl` package of the Zope framework. To declare class security information create a `ClassSecurityInfo` class attribute named `security`. The name `security` is used for consistency and for the benefit of new component authors, who often learn from looking at other people's code. You do not have to use the name `security` for the security infrastructure to recognize your assertion information, but it is recommended as a convention. For example:

```
from AccessControl import ClassSecurityInfo

class Mailbox(ObjectManager):
  """A mailbox object that contains mail message objects."""

  # Create a SecurityInfo for this class. We will use this
  # in the rest of our class definition to make security
  # assertions.
  security = ClassSecurityInfo()

  # Here is an example of a security assertion. We are
  # declaring that access to messageCount is public.
  security.declarePublic('messageCount')

  def messageCount(self):
    """Return a count of messages."""
    return len(self._messages)
```

Note that in the example above we called the `declarePublic` method of the `ClassSecurityInfo` instance to declare that access to the `messageCount` method be public. To make security assertions for your object, you just call the appropriate methods of the

`ClassSecurityInfo` object, passing the appropriate information for the assertion you are making.

`ClassSecurityInfo` approach has a number of benefits. A major benefit is that it is very explicit, it allows your security assertions to appear in your code near the objects they protect, which makes it easier to assess the state of protection of your code at a glance. The `ClassSecurityInfo` interface also allows you as a component developer to ignore the implementation details in the security infrastructure and protects you from future changes in those implementation details.

Let's expand on the example above and see how to make the most common security assertions using the `SecurityInfo` interface.

To assert that a method is *public* (anyone may call it) you may call the `declarePublic` method of the `SecurityInfo` object, passing the name of the method or subobject that you are making the assertion on:

```
security.declarePublic(methodName)
```

To assert that a method is *private* you call the `declarePrivate` method of the `SecurityInfo` object, passing the name of the method or subobject that you are making the assertion on:

```
security.declarePrivate(methodName)
```

To assert that a method or subobject is *protected* by a particular permission, you call the `declareProtected` method of the `SecurityInfo` object, passing a permission name and the name of a method to be protected by that permission:

```
security.declareProtected(permissionName, methodName)
```

If you have lots of methods you want to protect under the same permission, you can pass as many methodNames ase you want:

```
security.declareProtected(permissionName, methodName1,
methodName2, methodName3, ...)
```

Passing multiple names like this works for all of the `declare` security methods ('declarePublic', `declarePrivate`, and 'declareProtected').

Deciding To Use `declareProtected` vs. `declarePublic` or `declarePrivate`

If the method you're making the security declaration against is innocuous, and you're confident that its execution will not disclose private information nor make inappropriate changes to system state, you should declare the method public.

If a method should never be run under any circumstances via traversal or via through−the−web code, the method should be declared private. This is the default if a method has no security assertion, so you needn't explicitly protect unprotected methods unless you've used `setDefaultAccess` to set the object's default access policy to `allow` (detailed in *Other Assertions*, below).

If the method should only be executable by a certain class of users, you should declare the method protected.

A Class Security Example

Let's look at an expanded version of our `Mailbox` example that makes use of each of these types of security assertions:

```
from AccessControl import ClassSecurityInfo
import Globals

class Mailbox(ObjectManager):
  """A mailbox object."""

# Create a SecurityInfo for this class
  security = ClassSecurityInfo()

  security.declareProtected('View management screens', 'manage')
  manage=HTMLFile('mailbox_manage', globals())

  security.declarePublic('messageCount')
  def messageCount(self):
    """Return a count of messages."""
    return len(self._messages)

  # protect 'listMessages' with the 'View Mailbox' permission
  security.declareProtected('View Mailbox', 'listMessages')

  def listMessages(self):
    """Return a sequence of message objects."""
    return self._messages[:]

  security.declarePrivate('getMessages')
  def getMessages(self):
    self._messages=GoGetEm()
    return self._messages

# call this to initialize framework classes, which
# does the right thing with the security assertions.
Globals.InitializeClass(Mailbox)
```

Note the last line in the example. In order for security assertions to be correctly applied to your class, you must call the global class initializer for all classes that have security information. This is very important − the global initializer does the dirty work required to ensure that your object is protected correctly based on the security assertions that you have made. The initializer can be treated as a "black box" by the programmer − its will take care of protecting things correctly based on your security assertions. The global class initializer is located in the `Globals` module.

Deciding Permission Names For Protected Methods

When possible, you should make use of an existing Zope permission within a `declareProtected` assertion. A list of the permissions which are available in a default Zope installation is available within Appendix A. When it's not possible to reuse an existing permission, you should choose a permission name which is a verb or a verb phrase.

Object Assertions

Often you will also want to make a security assertion on the *object itself*. This is important for cases where your objects may be accessed in a restricted environment such as DTML. Consider the example DTML code:

```
<dtml-var expr="some_method(someObject)">
```

Here we are trying to call `some_method`, passing the object `someObject`. When this is evaluated in the restricted DTML environment, the security policy will attempt to validate access to both `some_method` and `someObject`. We've seen how to make assertions on methods − but in the case of `someObject` we are not trying to access any particular method, but rather the *object itself* (to pass it to 'some_method'). Because the security machinery will try to validate access to `someObject`, we need a way to let the security machinery know how to handle access to the object itself in addition to protecting its methods.

To make security assertions that apply to the *object itself* you call methods on the `SecurityInfo` object that are analogous to the three that we have already seen:

```
security.declareObjectPublic()

security.declareObjectPrivate()

security.declareObjectProtected(permissionName)
```

The meaning of these methods is the same as for the method variety, except that the assertion is made on the object itself.

An Object Assertion Example

Here is the updated `Mailbox` example, with the addition of a security assertion that protects access to the object itself with the `View Mailbox` permission:

```
from AccessControl import ClassSecurityInfo
import Globals

class Mailbox(ObjectManager):
  """A mailbox object."""

  # Create a SecurityInfo for this class
  security = ClassSecurityInfo()

  # Set security for the object itself
  security.declareObjectProtected('View Mailbox')

  security.declareProtected('View management screens', 'manage')
  manage=HTMLFile('mailbox_manage', globals())

  security.declarePublic('messageCount')
  def messageCount(self):
    """Return a count of messages."""
    return len(self._messages)

  # protect 'listMessages' with the 'View Mailbox' permission
  security.declareProtected('View Mailbox', 'listMessages')

  def listMessages(self):
    """Return a sequence of message objects."""
    return self._messages[:]

  security.declarePrivate('getMessages')
  def getMessages(self):
    self._messages=GoGetEm()
    return self._messages
```

```
        # call this to initialize framework classes, which
        # does the right thing with the security assertions.
        Globals.InitializeClass(Mailbox)
```

Other Assertions

The SecurityInfo interface also supports the less common security assertions noted earlier in this document.

To assert that access to subobjects that do not have explicit security information should be *allowed* rather than *denied* by the security policy, use:

```
        security.setDefaultAccess("allow")
```

This assertion should be used with caution. It will effectively change the access policy to "allow−by−default" for all attributes in your object instance (not just class attributes) that are not protected by explicit assertions. By default, the Zope security policy flatly denies access to attributes and methods which are not mentioned within a security assertion. Setting the default access of an object to "allow" effectively reverses this policy, allowing access to all attributes and methods which are not explicitly protected by a security assertion.

`setDefaultAccess` applies to attributes that are simple Python types as well as methods without explicit protection. This is important because some mutable Python types (lists, dicts) can then be modified by restricted code. Setting default access to "allow" also affects attributes that may be defined by the base classes of your class, which can lead to security holes if you are not sure that the attributes of your base classes are safe to access.

Setting the default access to "allow" should only be done if you are sure that all of the attributes of your object are safe to access, since the current architecture does not support using explicit security assertions on non−method attributes.

What Happens When You Make A Mistake Making `SecurityInfo` Declarations?

It's possible that you will make a mistake when making `SecurityInfo` declarations. For example, it is not legal to declare two conflicting permissions on a method:

```
        class Foo(SimpleItem):
            security = ClassSecurityInfo()

            meta_type='Foo'

            security.declareProtected('View foos', 'index_html')
            def index_html(self):
                """ make index_html web−publishable """
                return "<html><body>hi!</body></html>"

            security.declareProtected('View', 'index_html')
            # whoops, declared a conflicting permission on index_html!
```

When you make a mistake like this, the security machinery will accept the *first* declaration made in the code and will write an error to the Zope debug log upon encountering the second and following conflicting declarations during class initialization. It's similarly illegal to declare a method both private and public, or to declare a method both private and protected, or to declare a method both public and protected. A similar error will be raised in all of these cases.

Setting Default Roles For Permissions

When defining operations that are protected by permissions, one thing you commonly want to do is to arrange for certain roles to be associated with a particular permission *by default* for instances of your object.

For example, say you are creating a *News Item* object. You want `Anonymous` users to have the ability to view news items by default; you don't want the site manager to have to explicitly change the security settings for each *News Item* just to give the 'Anonymous" role `View` permission.

What you want as a programmer is a way to specify that certain roles should have certain permissions by default on instances of your object, so that your objects have sensible and useful security settings at the time they are created. Site managers can always *change* those settings if they need to, but you can make life easier for the site manager by setting up defaults that cover the common case by default.

As we saw earlier, the `SecurityInfo` interface provided a way to associate methods with permissions. It also provides a way to associate a permission with a set of default roles that should have that permission on instances of your object.

To associate a permission with one or more roles, use the following:

```
security.setPermissionDefault(permissionName, rolesList)
```

The *permissionName* argument should be the name of a permission that you have used in your object and *rolesList* should be a sequence (tuple or list) of role names that should be associated with *permissionName* by default on instances of your object.

Note that it is not always necessary to use this method. All permissions for which you did not set defaults using `setPermissionDefault` are assumed to have a single default role of `Manager`. Notable exceptions to this rule include `View` and `Access contents information`, which always have the default roles `Manager` and `Anonymous`.

The `setPermissionDefault` method of the `SecurityInfo` object should be called only once for any given permission name.

An Example of Associating Default Roles With Permissions

Here is our `Mailbox` example, updated to associate the `View Mailbox` permission with the roles `Manager` and `Mailbox Owner` by default:

```
from AccessControl import ClassSecurityInfo
import Globals

class Mailbox(ObjectManager):
  """A mailbox object."""

  # Create a SecurityInfo for this class
  security = ClassSecurityInfo()

  # Set security for the object itself
  security.declareObjectProtected('View Mailbox')

  security.declareProtected('View management screens', 'manage')
  manage=DTMLFile('mailbox_manage', globals())
```

```
      security.declarePublic('messageCount')
      def messageCount(self):
        """Return a count of messages."""
        return len(self._messages)

      security.declareProtected('View Mailbox', 'listMessages')
      def listMessages(self):
        """Return a sequence of message objects."""
        return self._messages[:]

      security.setPermissionDefault('View Mailbox', ('Manager', 'Mailbox Owner'))

    # call this to initialize framework classes, which
    # does the right thing with the security assertions.
    Globals.InitializeClass(Mailbox)
```

What Happens When You Make A Mistake Declaring Default Roles?

It's possible that you will make a mistake when making default roles declarations. For example, it is not legal to declare two conflicting default roles for a permission:

```
class Foo(SimpleItem):
    security = ClassSecurityInfo()

    meta_type='Foo'

    security.declareProtected('View foos', 'index_html')
    def index_html(self):
        """ """
        return "<html><body>hi!</body></html>"

    security.setPermissionDefault('View foos', ('Manager',))

    security.setPermissionDefault('View foos', ('Anonymous',))
    # whoops, conflicting permission defaults!
```

When you make a mistake like this, the security machinery will accept the *first* declaration made in the code and will write an error to the Zope debug log about the second and following conflicting declarations upon class initialization.

What Can (And Cannot) Be Protected By Class Security Info?

It is important to note what can and cannot be protected using the `ClassSecurityInfo` interface. First, the security policy relies on *Acquisition* to aggregate access control information, so any class that needs to work in the security policy must have either `Acquisition.Implicit` or `Acquisition.Explicit` in its base class hierarchy.

The current security policy supports protection of methods and protection of subobjects that are instances. It does *not* currently support protection of simple attributes of basic Python types (strings, ints, lists, dictionaries). For instance:

```
from AccessControl import ClassSecurityInfo
import Globals

# We subclass ObjectManager, which has Acquisition in its
# base class hierarchy, so we can use SecurityInfo.
```

```
class MyClass(ObjectManager):
  """example class"""

  # Create a SecurityInfo for this class
  security = ClassSecurityInfo()

  # Set security for the object itself
  security.declareObjectProtected('View')

  # This is ok, because subObject is an instance
  security.declareProtected('View management screens', 'subObject')
  subObject=MySubObject()

  # This is ok, because sayHello is a method
  security.declarePublic('sayHello')
  def sayHello(self):
    """Return a greeting."""
    return "hello!"

  # This will not work, because foobar is not a method
  # or an instance − it is a standard Python type
  security.declarePublic('foobar')
  foobar='some string'
```

Keep this in mind when designing your classes. If you need simple attributes of your objects to be accessible (say via DTML), then you need to use the `setDefaultAccess` method of `SecurityInfo` in your class to allow this (see the note above about the security implications of this). In general, it is always best to expose the functionality of your objects through methods rather than exposing attributes directly.

Note also that the actual `ClassSecurityInfo` instance you use to make security assertions is implemented such that it is *never* accessible from restricted code or through the Web (no action on the part of the programmer is required to protect it).

Inheritance And Class Security Declarations

Python inheritance can prove confusing in the face of security declarations.

If a base class which has already been run through "InitializeClass" is inherited by a superclass, nothing special needs to be done to protect the base class' methods within the superclass unless you wish to modify the declarations made in the base class. The security declarations "filter down" into the superclass.

On the other hand, if a base class hasn't been run through InitializeClass, you need to proxy its security declarations in the superclass if you wish to access any of its methods within through−the−web code or via URL traversal.

In other words, security declarations that you make using `ClassSecurityInfo` objects effect instances of the class upon which you make the declaration. You only need to make security declarations for the methods and subobjects that your class actually *defines*. If your class inherits from other classes, the methods of the base classes are protected by the security declarations made in the base classes themselves. The only time you would need to make a security declaration about an object defined by a base class is if you needed to *redefine* the security information in a base class for instances of your own class. An example below redefines a security assertion in a subclass:

```
from AccessControl import ClassSecurityInfo
```

```
import Globals

class MailboxBase(ObjectManager):
  """A mailbox base class."""

  # Create a SecurityInfo for this class
  security = ClassSecurityInfo()

  security.declareProtected('View Mailbox', 'listMessages')
  def listMessages(self):
    """Return a sequence of message objects."""
    return self._messages[:]

  security.setPermissionDefault('View Mailbox', ('Manager', 'Mailbox Owner'))

Globals.InitializeClass(MailboxBase)

class MyMailbox(MailboxBase):
  """A mailbox subclass, where we want the security for
    listMessages to be public instead of protected (as
    defined in the base class)."""

  # Create a SecurityInfo for this class
  security = ClassSecurityInfo()

  security.declarePublic('listMessages')

Globals.InitializeClass(Mailbox)
```

Class Security Assertions In Non−Product Code (External Methods/Python Scripts)

Objects that are returned from Python Scripts or External Methods need to have assertions declared for themselves before they can be used in restricted code. For example, assume you have an External Method that returns instances of a custom `Book` class. If you want to call this External Method from DTML, and you'd like your DTML to be able to use the returned `Book` instances, you will need to ensure that your class supports Acquisition, and you'll need to make security assertions on the `Book` class and initialize it with the global class initializer (just as you would with a class defined in a Product). For example:

```
# an external method that returns Book instances

from AccessControl import ClassSecurityInfo
from Acquistion import Implicit
import Globals

class Book(Implicit):

  def __init__(self, title):
    self._title=title

  # Create a SecurityInfo for this class
  security = ClassSecurityInfo()
  security.declareObjectPublic()

  security.declarePublic('getTitle')
  def getTitle(self):
    return self._title

Globals.InitializeClass(Book)
```

```
# The actual external method
def GetBooks(self):
  books=[]
  books.append(Book('King Lear'))
  books.append(Book('Romeo and Juliet'))
  books.append(Book('The Tempest'))
  return books
```

Note that this particular example is slightly dangerous. You need to be careful that classes defined in external methods not be made persistent, as this can cause Zope object database inconsistencies. In terms of this example, this would mean that you would need to be careful to not attach the Book object returned from the `GetBook` method to a persistent object within the ZODB. See Chapter 4, "ZODB Persistent Components" for more information. Thus it's generally a good idea to define the Book class in a Product if you want books to be persistent. It's also less confusing to have all of your security declarations in Products.

However, one benefit of the `SecurityInfo` approach is that it is relatively easy to subclass and add security info to classes that you did not write. For example, in an External Method, you may want to return instances of `Book` although `Book` is defined in another module out of your direct control. You can still use `SecurityInfo` to define security information for the class by using:

```
# an external method that returns Book instances

from AccessControl import ClassSecurityInfo
from Acquisition import Implicit
import bookstuff
import Globals

class Book(Implicit, bookstuff.Book):
  security = ClassSecurityInfo()
  security.declareObjectPublic()
  security.declarePublic('getTitle')

Globals.InitializeClass(Book)

# The actual external method
def GetBooks(self):
  books=[]
  books.append(Book('King Lear'))
  books.append(Book('Romeo and Juliet'))
  books.append(Book('The Tempest'))
  return books
```

Module Security Assertions

Another kind of `SecurityInfo` object you will use as a component developer is the `ModuleSecurityInfo` object.

`ModuleSecurityInfo` objects do for objects defined in modules what `ClassSecurityInfo` objects do for methods defined in classes. They allow module−level objects (generally functions) to be protected by security assertions. This is most useful when attempting to allow through−the−web code to `import` objects defined in a Python module.

One major difference between `ModuleSecurityInfo` objects and ClassSecurityInfo objects is that `ModuleSecurityInfo` objects cannot be declared `protected` by a permission. Instead, ModuleSecurityInfo objects may only declare that an object is `public` or `private`. This is due to the fact that modules are essentially "placeless", global things, while permission protection depends heavily on "place" within Zope.

Declaring Module Security

In order to use a filesystem Python module from restricted code such as Python Scripts, the module must have Zope security declarations associated with functions within it. There are a number of ways to make these declarations:

- By embedding the security declarations in the target module. A module that is written specifically for Zope may do so, whereas a module not specifically written for Zope may not be able to do so.
- By creating a wrapper module and embedding security declarations within it. In many cases it is difficult, impossible, or simply undesirable to edit the target module. If the number of objects in the module that you want to protect or make public is small, you may wish to simply create a wrapper module. The wrapper module imports objects from the wrapped module and provides security declarations for them.
- By placing security declarations in a filesystem Product. Filesystem Python code, such as the `__init__.py` of a Product, can make security declarations on behalf of an external module. This is also known as an "external" module security info declaration.

The `ModuleSecurityInfo` class is defined in the `AccessControl` package of the Zope framework.

Using ModuleSecurityInfo Objects

Instances of `ModuleSecurityInfo` are used in two different situations. In embedded declarations, inside the module they affect. And in external declarations, made on behalf of a module which may never be imported.

Embedded ModuleSecurityInfo Declarations

An embedded ModuleSecurityInfo declaration causes an object in its module to be importable by through−the−web code.

Here's an example of an embedded declaration:

```
from AccessControl import ModuleSecurityInfo
modulesecurity = ModuleSecurityInfo()
modulesecurity.declarePublic('foo')

def foo():
    return "hello"
    # foo

modulesecurity.apply(globals())
```

When making embedded ModuleSecurityInfo declarations, you should instantiate a ModuleSecurityInfo object and assign it to a name It's wise to use the recommended name `modulesecurity` for consistency's sake. You may then use the modulesecurity object's `declarePublic` method to declare functions inside of the current module as public. Finally, appending the last line ("modulesecurity.apply(globals())") is an important step. It's necessary in order to poke the security machinery into action. The above example declares the `foo` function public.

The name `modulesecurity` is used for consistency and for the benefit of new component authors, who often learn from looking at other people's code. You do not have to use the name `modulesecurity` for the security infrastructure to recognize your assertion information, but it is recommended as a convention.

External ModuleSecurityInfo Declarations

By creating a ModuleSecurityInfo instance with a module name argument, you can make declarations on behalf of a module without having to edit or import the module.

Here's an example of an external declaration:

```
from AccessControl import ModuleSecurityInfo
# protect the 'foo' function within (yet−to−be−imported) 'foomodule'
ModuleSecurityInfo('foomodule').declarePublic('foo')
```

This declaration will cause the following code to work within PythonScripts:

```
from foomodule import foo
```

When making external ModuleSecurityInfo declarations, you needn't use the "modulesecurity.apply(globals())" idiom demonstrated in the embedded declaration section above. As a result, you needn't assign the ModuleSecurityInfo object to the name `modulesecurity`.

Providing Access To A Module Contained In A Package

Note that if you want to provide access to a module inside of a package which lives in your PYTHONPATH, you'll need to provide security declarations for *all of the the packages and sub−packages along the path used to access the module.*

For example, assume you have a function foo, which lives inside a module named `module`, which lives inside a package named `package2`, which lives inside a package named `package1` You might declare the `foo` function public via this chain of declarations:

```
ModuleSecurityInfo('package1').declarePublic('package2')
ModuleSecurityInfo('package1.package2').declarePublic('module')
ModuleSecurityInfo('package1.package2.module').declarePublic('foo')
```

Note that in the code above we took the following steps:

- make a ModuleSecurityInfo object for `package1`
- call the declarePublic method of the `package1` ModuleSecurityInfo object, specifying `package2` as what we're declaring public. This allows through the web code to "see" package2 inside package1.
- make a ModuleSecurityInfo object for `package1.package2`.
- call the declarePublic method of the `package1.package2` ModuleSecurityInfo object, specifying `module` as what we're declaring public. This allows through the web code to "see" `package1.package2.module`.
- declare `foo` public inside the ModuleSecurityInfo for `package1.package2.module`.

Through−the−web code may now perform an import ala: 'import package1.package2.module.foo'

Many people who use Zope will be concerned with using ModuleSecurityInfo to make declarations on modules which live within Zope's Products directory. This is just an example of declaring module security on a module within a package. Here is an example of using ModuleSecurityInfo to make security declarations on behalf of the `CatalogError` class in the `ZCatalog.py` module. This could be placed, for instance, within the any Product's `__init__.py` module:

```
from AccessControl import ModuleSecurityInfo
ModuleSecurityInfo('Products').declarePublic('Catalog')
ModuleSecurityInfo('Products.Catalog').declarePublic('CatalogError')
```

Declaring Module Security On Modules Implemented In C

Certain modules, such as the standard Python `sha` module, provide extension types instead of classes, as the `sha` module is implemented in C. Security declarations typically cannot be added to extension types, so the only way to use this sort of module is to write a Python wrapper class, or use External Methods.

Default Module Security Info Declarations

Through−the−web Python Scripts are by default able to import a small number of Python modules for which there are security declarations. These include `string`, `math`, and `random`. The only way to make other Python modules available for import is to add security declarations to them in the filesystem.

Utility Functions For Allowing Import of Modules By Through The Web Code

Instead of manually providing security declarations for each function in a module, the utility function "allow_class" and "allow_module" have been created to help you declare the entire contents of a class or module as public.

You can handle a module, such as base64, that contains only safe functions by writing `allow_module("module_name")`. For instance:

```
from Products.PythonScripts.Utility import allow_module
allow_module("base64")
```

This statement declares all functions in the `base64` module (`encode`, `decode`, `encodestring`, and `decodestring`) as public, and from a script you will now be able to perform an import statement such as "from base64 import encodestring".

To allow access to only some names in a module, you can eschew the allow_class and allow_module functions for the lessons you learned in the previous section and do the protection "manually":

```
from AccessControl import ModuleSecurityInfo
ModuleSecurityInfo('module_name').declarePublic('name1','name2', ...)
```

Making Permission Assertions On A Constructor

When you develop a Python disk−based product, you will generally be required to make "constructor" methods for the objects which you wish to make accessible via the Zope management interface by users of your Product. These constructors are usually defined within the modules which contain classes which are intended to be turned into Zope instances. For more information on how constructors are used in Zope with security, see Chapter 3 "Zope Products".

The Zope Product machinery "bootstraps" Product−based classes with proper constructors into the namespace of the Zope management interface "Add" list at Zope startup time. This is done as a consequence of registering a class by way of the Product's `__init__.py intialize` function. If you want to make, for example, the imaginary `FooClass` in your Product available from the "Add" list, you may construct an `__init__.py` file that looks much like this:

```
from FooProduct import FooClass

def initialize(context):
    """ Initialize classes in the FooProduct module """
    context.registerClass(
        FooProduct.FooClass, # the class object
        permission='Add FooClasses',
        constructors=(FooProduct.manage_addFooClassForm,
                      FooProduct.manage_addFooClass),
        icon='foo.gif'
        )
```

The line of primary concern to us above is the one which says "permission='Add FooClasses'". This is a permission declaration which, thanks to Zope product initialization, restricts the adding of FooClasses to those users who have the `Add FooClasses` permission by way of a role association determined by the system administrator.

If you do not include a `permission` argument to `registerClass`, then Zope will create a default permission named 'Add [meta−type]s'. So, for example, if your object had a meta_type of `Animal`, then Zope would create a default permission, 'Add Animals'. For the most part, it is much better to be explicit then to rely on Zope to take care of securty details for you, so be sure to specify a permission for your object.

Designing For Security

"Security is hard."
Jim Fulton.

When you're under a deadline, and you "just want it to work", dealing with security can be difficult. As a component developer, following these basic guidelines will go a long way toward avoiding problems with security integration. They also make a good debugging checklist!

- Ensure that any class that needs to work with security has `Acquisition.Implicit` or `Acquisition.Explicit` somewhere in its base class hierarchy.
- Design the interface to your objects around methods; don't expect clients to access instance attributes directly.
- Ensure that all methods meant for use by restricted code have been protected with appropriate security assertions.
- Ensure that you called the global class initializer on all classes that need to work with security.

Compatibility

The implementation of the security assertions and `SecurityInfo` interfaces described in this document are available in Zope 2.3 and higher.

Zope components that do not use the new `SecurityInfo` interfaces for security assertions (ones which use older mechanisms) will continue to work without modification until further notice.

Using The RoleManager Base Class With Your Zope Product

After your Product is deployed, system managers and other users of your Product often must deal with security settings on instances they make from your classes.

Product classes which inherit Zope's standard RoleManager base class allow instances of the class to present a security interface. This security interface allows managers and developers of a site to control an instance's security settings via the Zope management interface.

The user interface is exposed via the *Security* management view. From this view, a system administrator may secure instances of your Product's class by associating roles with permissions and by asserting that your object instance contains "local roles". It also allows them to create "user−defined roles" within the Zope management framework in order to associate these roles with the permissions of your product and with users. This user interface and its usage patterns are explained in more detail within the Zope Book's security chapter.

If your Product's class does not inherit from `RoleManager`, its methods will still retain the security assertions associated with them, but you will be unable to allow users to associate roles with the permissions you've defined respective to instances of your class. Your objects will also not allow local role definitions. Note that objects which inherit from the `SimpleItem.SimpleItem` mixin class already inherit from `RoleManager`.

Conclusion

Zope security is based upon roles and permissions. Users have roles. Security policies map permissions to roles. Classes protect methods with permissions. As a developer you main job is to protect your classes by associating methods with permissions. Of course there are many other details such as protecting modules and functions, creating security user interfaces, and initializing security settings.

Chapter 7: Testing and Debugging

As you develop Zope applications you will run into problems. This chapter covers debugging and testing techniques that can help you. The Zope debugger allow you to peek inside a running process and find exactly what is going wrong. Unit testing allows you to automate the testing process to ensure that your code still works correctly as you change it. Finally, Zope provides logging facilities which allow you to emit warnings and error messages.

Debugging

Zope provides debugging information through a number of sources. It also allows you a couple avenues for getting information about Zope as it runs.

The Control Panel

The control panel provides a number of views that can help you debug Zope, especially in the area of performance. The *Debugging Information* link on the control panel provides two views, *Debugging Info* and *Profiling*.

Debugging info provides information on the number of object references and the status of open requests. The object references list displays the name of the object and the number of references to that object in Zope. Understanding how reference counts help debugging is a lengthy subject, but in general you can spot memory leaks in your application if the number of references to certain objects increases without bound. The busier your site is, or the more content it holds, the more reference counts you will tend to have.

Profiling uses the standard Python profiler. This is turned on by setting the `PROFILE_PUBLISHER` environment variable before executing Zope.

When the profiler is running, the performance of your Zope system will suffer a lot. Profiling should only be used for short periods of time, or on a separate ZEO client so that your normal users to not experience this significant penalty.

Profiling provides you with information about which methods in your Zope system are taking the most time to execute. It builds a *profile*, which lists the busiest methods on your system, sorted by increasing resource usage. For details on the meaning of the profiler's output, read the standard Python documentation.

Product Refresh Settings

As of Zope 2.4 there is a *Refresh* view on all Control Panel Products. Refresh allows you to reload your product's modules as you change them, rather than having to restart Zope to see your changes. The *Refresh* view provides the same debugging functionality previously provided by Shane Hathaway's Refresh Product.

To turn on product refresh capabilities place a `refresh.txt` file in your product's directory. Then visit the *Refresh* view of your product in the management interface. Here you can manually reload your product's modules with the *Refresh this product* button. This allows you to immediately see the effect of your changes, without restarting Zope. You can also turn on automatic refreshing which causes Zope to frequently check for changes to your modules and refresh your product when it detects that your files have changed. Since automatic refresh causes Zope to run more slowly, it is a good idea to only turn it on for a few products at a time.

Debug Mode

Setting the `Z_DEBUG_MODE=1` environment puts Zope into debug mode. This mode reduces the performance of Zope a little bit. Debug model has a number of wide ranging effects:

- Tracebacks are shown on the browser when errors are raised.
- External Methods and DTMLFile objects are checked to see if they have been modified every time they are called. If modified, they are reloaded.
- Zope will not fork into the background in debug mode, instead, it will remain attached to the terminal that started it and the main logging information will be redirected to that terminal.

Normally, debug mode is set using the `−D` switch when starting Zope, though you can set the environment variable directly if you wish.

By using debug mode and product refresh together you will have little reason to restart Zope while developing.

The Python Debugger

Zope is integrated with the Python debugger (pdb). The Python debugger is pretty simple as command line debuggers go, and anyone familiar with other popular command line debuggers (like gdb) will feel right at home in pdb.

For an introduction to pdb see the standard pdb documentation.

There are a number of ways to debug a Zope process:

- You can shut down the Zope server and simulate a request on the command line.
- You can run a special ZEO client that debugs a running server.
- You can run Zope in debug model and enter the debugger through Zope's terminal session.

The first method is an easy way to debug Zope if you are not running ZEO. First, you must first shut down the Zope process. It is not possible to debug Zope in this way and run it at the same time. Starting up the debugger this way will by default start Zope in single threaded mode.

For most Zope developer's purposes, the debugger is needed to debug some sort of application level programming error. A common scenario is when developing a new product for Zope. Products extend Zope's functionality but they also present the same kind of debugging problems that are commonly found in any programming environment. It is useful to have an existing debugging infrastructure to help you jump immediately to your new object and debug it and play with it directly in pdb. The Zope debugger lets you do this.

In reality, the "Zope" part of the Zope debugger is actually just a handy way to start up Zope with some pre−configured break points and to tell the Python debugger where in Zope you want to start debugging.

Simulating HTTP Requests

Now for an example. Remember, for this example to work, you *must* shut down Zope. Go to your Zope's `lib/python` directory and fire up Python and import `Zope` and `ZPublisher`:

```
$ cd lib/python
$ python
Python 1.5.2 (#0, Apr 13 1999, 10:51:12) [MSC 32 bit (Intel)] on win32
Copyright 1991−1995 Stichting Mathematisch Centrum, Amsterdam
>>> import Zope, ZPublisher
>>>
```

Here you have run the Python interpreter (which is where using the debugger takes place) and imported two modules, `Zope` and `ZPublisher`. If Python complains about an `ImportError` and not being able to find either module, then you are probably in the wrong directory, or you have not compiled Zope properly. If you get this message:

```
ZODB.POSException.StorageSystemError: Could not lock the
database file. There must be another process that has opened
the file.
```

This tells you that Zope is currently running. Shutdown Zope and try again.

The `Zope` module is the main Zope application module. When you import `Zope` it sets up Zope. `ZPublisher` is the Zope ORB. See Chapter 2 for more information about `ZPublisher`.

You can use the `ZPublisher.Zope` function to simulate an HTTP request. Pass the function a URL relative the your root Zope object. Here is an example of how to simulate an HTTP request from the debugger:

```
>>> ZPublisher.Zope('')
Status: 200 OK
X−Powered−By: Zope (www.zope.org), Python (www.python.org)
Content−Length: 1238
Content−Type: text/html

<HTML><HEAD><TITLE>Zope</TITLE>

  ... blah blah...

</BODY></HTML>
>>>
```

If you look closely, you will see that the content returned is *exactly* what is returned when you call your root level object through HTTP, including all the HTTP headers.

Keep in mind that calling Zope this way does NOT involve a web server. No ports are opened, the `ZServer` code is not even imported. In fact, this is just an interpreter front end to the same application code the ZServer *does* call.

Interactive Debugging

Debugging involves publishing a request up to a point where you think it's failing, and then inspecting the state of your variables and objects. The easy part is the actual inspection, the hard part is getting your program to stop at the right point.

So, for the sake our example, let's say that you have a `News` object which is defined in a Zope Product called `ZopeNews`, and is located in the `lib/python/Products/ZopeNews` directory. The class that defines the `News` instance is also called `News`, and is defined in the `News.py` module in your product.

Therefore, from Zope's perspective the fully qualified name of your class is `Products.ZopeNews.News.News`. All Zope objects have this kind of fully qualified name. For example, the `ZCatalog` class can be found in `Products.ZCatalog.ZCatalog.ZCatalog` (The redundancy is because the product, module, and class are all named 'ZCatalog').

Now let's create an example method to debug. You want your news object to have a `postnews` method, that posts news:

```
class News(...):

    ...

    def postnews(self, news, author="Anonymous"):
        self.news = news

    def quote(self):
        return '%s said, "%s"' % (self.author, self.news)
```

You may notice that there's something wrong with the `postnews` method. The method assigns `news` to an instance variable, but it does nothing with `author`. If the `quote` method is called, it will raise an `AttributeError` when it tries to look up the name `self.author`. Although this is a pretty obvious goof, we'll use it to illustrate using the debugger to fix it.

Running the debugger is done in a very similar way to how you called Zope through the python interpreter before, except that you introduce one new argument to the call to `Zope`:

```
>>> ZPublisher.Zope('/News/postnews?new=blah', d=1)
* Type "s<cr>c<cr>" to jump to beginning of real publishing process.
* Then type c<cr> to jump to the beginning of the URL traversal
  algorithm.
* Then type c<cr> to jump to published object call.
> <string>(0)?()
pdb>
```

Here, you call Zope from the interpreter, just like before, but there are two differences. First, you call the `postnews` method with an argument using the URL, `/News/postnews?new=blah`. Second, you provided a new argument to the Zope call, `d=1`. The `d` argument, when true, causes Zope to fire up in the Python debugger, pdb. Notice how the Python prompt changed from `>>>` to `pdb>`. This indicates that you are in the debugger.

When you first fire up the debugger, Zope gives you a helpful message that tells you how to get to your object. To understand this message, it's useful to know how you have set Zope up to be debugged. When Zope fires up in debugger mode, there are three breakpoints set for you automatically (if you don't know what a breakpoint is, you need to read the python debugger documentation.).

The first breakpoint stops the program at the point that ZPublisher (the Zope ORB) tries to publish the application module (in this case, the application module is 'Zope'). The second breakpoint stops the program right before ZPublisher tries to traverse down the provided URL path (in this case, '/News/postnews'). The third breakpoint will stop the program right before ZPublisher calls the object it finds that matches the URL path (in this case, the `News` object).

So, the little blurb that comes up and tells you some keys to press is telling you these things in a terse way. Hitting `s` will *step* you into the debugger, and hitting `c` will *continue* the execution of the program until it hits a breakpoint.

Note however that none of these breakpoints will stop the program at `postnews`. To stop the debugger right there, you need to tell the debugger to set a new breakpoint. Why a new breakpoint? Because Zope will stop you before it traverse your objects path, it will stop you before it calls the object, but if you want to stop it *exactly* at some point in your code, then you have to be explicit. Sometimes the first three breakpoints are convienent, but often you need to set your own special break point to get you exactly where you want to go.

Setting a breakpoint is easy (and see the next section for an even easier method). For example:

```
pdb> import Products
pdb> b Products.ZopeNews.News.News.postnews
Breakpoint 5 at C:\Program Files\WebSite\lib\python\Products\ZopeNews\News.py:42
pdb>
```

First, you import `Products`. Since your module is a Zope product, it can be found in the `Products` package. Next, you set a new breakpoint with the *break* debugger command (pdb allows you to use single letter commands, but you could have also used the entire word 'break'). The breakpoint you set is `Products.ZopeNews.News.News.postnews`. After setting this breakpoint, the debugger will respond that it found the method in question in a certain file, on a certain line (in this case, the fictitious line 42) and return you to the debugger.

Now, you want to get to your `postnews` method so you can start debugging it. But along the way, you must first *continue* through the various breakpoints that Zope has set for you. Although this may seem like a bit of a burden, it's actually quite good to get a feel for how Zope works internally by getting down the rhythm that Zope uses to publish your object. In these next examples, my comments will begin with '#". Obviously, you won't see these comments when you are debugging. So let's debug:

```
pdb> s
# 's'tep into the actual debugging

> <string>(1)?()
# this is pdb's response to being stepped into, ignore it

pdb> c
# now, let's 'c'ontinue onto the next breakpoint

> C:\Program Files\WebSite\lib\python\ZPublisher\Publish.py(112)publish()
-> def publish(request, module_name, after_list, debug=0,

# pdb has stopped at the first breakpoint, which is the point where
# ZPubisher tries to publish the application module.

pdb> c
# continuing onto the next breakpoint you get...

> C:\Program Files\WebSite\lib\python\ZPublisher\Publish.py(101)call_object()
-> def call_object(object, args, request):
```

Here, `ZPublisher` (which is now publishing the application) has found your object and is about to call it. Calling your object consists of applying the arguments supplied by `ZPublisher` to the object. Here, you can see how `ZPublisher` is passing three arguments into this process. The first argument is `object` and is the actual object you want to call. This can be verified by *printing* the object:

```
pdb> p object
<News instance at 00AFE410>
```

Now you can inspect your object (with the *print* command) and even play with it a bit. The next argument is `args`. This is a tuple of arguments that `ZPublisher` will apply to your object call. The final argument is `request`. This is the request object and will eventually be transformed in to the DTML usable object `REQUEST`. Now continue, your breakpoint is next:

```
pdb> c
> C:\Program Files\WebSite\lib\python\Products\ZopeNews\News.py(42)postnews()
−> def postnews(self, N)
```

Now you are here, at your method. To be sure, tell the debugger to show you where you are in the code with the `l` command. Now you can examine variable and perform all the debugging tasks that the Python debugger provides. From here, with a little knowledge of the Python debugger, you should be able to do any kind of debugging task that is needed.

Interactive Debugging Triggered From the Web

If you are running in debug mode you can set break points in your code and then jump straight to the debugger when Zope comes across your break points. Here's how to set a breakpoint:

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

Now start Zope in debug mode and point your web browser at a URL that causes Zope to execute the method that includes a breakpoint. When this code is executed, the Python debugger will come up in the terminal where you started Zope. Also note that from your web browser it looks like Zope is frozen. Really it's just waiting for you do your debugging.

From the terminal you are inside the debugger as it is executing your request. Be aware that you are just debugging one thread in Zope, and other requests may be being served by other threads. If you go to the *Debugging Info* screen while in the debugger, you can see your debugging request and how long it has been open.

It is often more convenient to use this method to enter the debugger than it is to call `ZPublisher.Zope` as detailed in the last section.

Post−Mortem Debugging

Often, you need to use the debugger to chase down obscure problems in your code, but sometimes, the problem is obvious, because an exception gets raised. For example, consider the following method on your `News` class:

```
def quote(self):
    return '%s said, "%s"' % (self.Author, self.news)
```

Here, you can see that the method tries to substitute `self.Author` in a string, but earlier we saw that this should really be `self.author`. If you tried to run this method from the command line, an exception would be raised:

```
>>> ZPublisher.Zope('/News/quote')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "./News.py", line 4, in test
    test2()
  File "./News.py", line 3, in test2
    return '%s said, "%s"' % (self.Author, self.news)
NameError: Author
>>>
```

Using Zope's normal debugging methods, you would typically need to start from the "beginning" and step your way down through the debugger to find this error (in this case, the error is pretty obvious, but more often than not errors can be pretty obscure!).

Post−mortem debugging allows you to jump *directly* to the spot in your code that raised the exception, so you do not need to go through the possibly tedious task of stepping your way through a sea of Python code. In the case of our example, you can just pass `ZPublisher.Zope` call a `pm` argument that is set to 1:

```
>>> ZPublisher.Zope('/News/quote', pm=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "./News.py", line 4, in test
    test2()
  File "./News.py", line 3, in test2
    return '%s said, "%s"' % (self.Author, self.news)
NameError: Author
(pdb)
```

Here, you can see that instead of taking you back to a python prompt, the post mortem debugging flag has caused you to go right into the debugging, *exactly* at the point in your code where the exception is raised. This can be verified with the debugger's (l)ist command. Post mortem debugging offers you a handy way to jump right to the section of your code that is failing in some obvious way by raising an exception.

Debugging With ZEO

ZEO presents some interesting debugging abilities. ZEO lets you debug one ZEO client when other clients continue to server requests for your site. In the above examples, you have to shut down Zope to run in the debugger, but with ZEO, you can debug a production site while other clients continue to serve requests. Using ZEO is beyond the scope of this chapter. However, once you have ZEO running, you can debug a client process exactly as you debug a single−process Zope.

Unit Testing

Unit testing allows you to automatically test your classes to make sure they are working correctly. By using unit tests you can make sure as you develop and change your classes that you are not breaking them. Zope comes with Pyunit. You can find out more information on Pyunit at the Pyunit home page. Pyunit is also part of the Python standard library as of Python 2.1.

What Are Unit Tests

A "unit" may be defined as a piece of code with a single intended purpose. A "unit test" is defined as a piece of code which exists to codify the intended behavior of a unit and to compare its intended behavior against its actual behavior.

Unit tests are a way for developers and quality assurance engineers to quickly ascertain whether independent units of code are working as expected. Unit tests are generally written at the same time as the code they are intended to test. A unit testing framework allows a collection of unit tests to be run without human intervention, producing a minimum of output if all the tests in the collection are successful.

It's a good idea to have a sense of the limits of unit testing. From the Extreme Programming Enthusiast website here is a list of things that unit tests are *not*:

- Manually operated.
- Automated screen−driver tests that simulate user input (these are "functional tests").
- Interactive. They run "no questions asked."
- Coupled. They run without dependencies except those native to the thing being tested.
- Complicated. Unit test code is typically straightforward procedural code that simulates an event.

Writing Unit Tests

Here are the times when you should write unit tests:

- When you write new code
- When you change and enhance existing code
- When you fix bugs

It's much better to write tests when you're working on code than to wait until you're all done and then write tests.

You should write tests that exercise discrete "units" of functionality. In other words, write simple, specific tests that test one capability. A good place to start is with interfaces and classes. Classes and especially interfaces already define units of work which you may wish to test.

Since you can't possibly write tests for every single capability and special case, you should focus on testing the riskiest parts of your code. The riskiest parts are those that would be the most disastrous if they failed. You may also want to test particularly tricky or frequently changed things.

Here's an example test script that tests the `News` class defined earlier in this chapter:

```
import unittest
import News

class NewsTest(unittest.TestCase):

    def testPost(self):
        n=News()
        s='example news'
        n.postnews(s)
        assert n.news==s

    def testQuote(self):
        n=News()
        s='example news'
        n.postnews(s)
        assert n.quote()=='Anonymous said: "%s"' % s
        a='Author'
        n.postnews(s, a)
        assert n.quote()=='%s said: "%s"' % (a, s)
```

```
def test_suite():
   return unittest.makeSuite(NewsTest, 'news test')

def main():
   unittest.TextTestRunner().run(test_suite())

if __name__=="__main__":
    main()
```

You should save tests inside a `tests` sub−directory in your product's directory. Test scripts file names should start with test, for example `testNews.py`. You may accumulate many test scripts in your product's `tests` directory. You can run test your product by running the test scripts.

We cannot cover all there is to say about unit testing here. Take a look at the Pyunit documentation for more background on unit testing.

Zope Test Fixtures

One issue that you'll run into when unit testing is that you may need to set up a Zope environment in order to test your products. You can solve this problem in two ways. First, you can structure your product so that much of it can be tested without Zope (as you did in the last section). Second, you can create a test fixture that sets up a Zope environment for testing.

To create a test fixture for Zope you'll need to:

1. Add Zope's `lib/python` directory to the Python path.
2. Import `Zope` and any other needed Zope modules and packages.
3. Get a Zope application object.
4. Do your test using the application object.
5. Clean up the test by aborting or committing the transaction and closing the Zope database connection.

Here's an example Zope test fixture that demonstrates how to do each of these steps:

```
import os, os.path, sys, string
try:
    import unittest
except ImportError:
    fix_path()
    import unittest

class MyTest(unittest.TestCase):

    def setup(self):
        # Get the Zope application object and store it in an
        # instance variable for use by test methods
        import Zope
        self.app=Zope.app()

    def tearDown(self):
        # Abort the transaction and shut down the Zope database
        # connection.
        get_transaction().abort()
        self.app._p_jar.close()

    # At this point your test methods can perform tests using
```

```
        # self.app which refers to the Zope application object.

        ...

    def fix_path():
        # Add Zope's lib/python directory to the Python path
        file=os.path.join(os.getcwd(), sys.argv[0])
        dir=os.path.join('lib', 'python')
        i=string.find(file, dir)
        sys.path.insert(0, file[:i+len(dir)])

    def test_suite():
       return unittest.makeSuite(MyTest, 'my test')

    def main():
       unittest.TextTestRunner().run(test_suite())

    if __name__=="__main__":
        fix_path()
        main()
```

This example shows a fairly complete Zope test fixture. If your Zope tests only needs to import Zope modules and packages you can skip getting a Zope application object and closing the database transaction.

Some times you may run into trouble if your test assuming that there is a current Zope request. There are two ways to deal with this. One is to use the `makerequest` utility module to create a fake request. For example:

```
    class MyTest(unittest.TestCase):
        ...

        def setup(self):
            import Zope
            from Testing import makerequest
            self.app=makerequest.makerequest(Zope.app())
```

This will create a Zope application object that is wrapped in a request. This will enable code that expects to acquire a `REQUEST` attribute work correctly.

Another solution to testing methods that expect a request is to use the `ZPublisher.Zope` function described earlier. Using this approach you can simulate HTTP requests in your unit tests. For example:

```
    import ZPublisher

    class MyTest(unittest.TestCase):
        ...

        def testWebRequest(self):
            ZPublisher.Zope('/a/url/representing/a/method?with=a&couple=arguments',
                            u='username:password',
                            s=1,
                            e={'some':'environment', 'variable':'settings'})
```

If the `s` argument is passed to `ZPublisher.Zope` then no output will be sent to `sys.stdout`. If you want to capture the output of the publishing request and compare it to an expected value you'll need to do something like this:

```
    f=StringIO()
    temp=sys.stdout
```

```
        sys.stdout=f
        ZPublisher.Zope('/myobject/mymethod')
        sys.stdout=temp
        assert f.getvalue() == expected_output
```

Here's a final note on unit testing with a Zope test fixture: you may find Zope helpful. ZEO allows you to test an application while it continues to serve other users. It also speeds Zope start up time which can be a big relief if you start and stop Zope frequently while testing.

Despite all the attention we've paid to Zope testing fixtures, you should probably concentrate on unit tests that don't require a Zope test fixture. If you can't test much without Zope there is a good chance that your product would benefit from some refactoring to make it simpler and less dependent on the Zope framework.

Logging

Zope provides a framework for logging information to Zope's application log. You can configure Zope to write the application log to a file, syslog, or other back−end.

The logging API defined in the `zLOG` module. This module provides the `LOG` function which takes the following required arguments:

subsystem
: The subsystem generating the message (e.g. "ZODB")

severity
: The "severity" of the event. This may be an integer or a floating point number. Logging back ends may consider the int() of this value to be significant. For example, a back−end may consider any severity whose integer value is WARNING to be a warning.

summary
: A short summary of the event

These arguments to the `LOG` function are optional:

detail
: A detailed description

error
: A three−element tuple consisting of an error type, value, and traceback. If provided, then a summary of the error is added to the detail.

reraise
: If provided with a true value, then the error given by error is reraised.

You can use the `LOG` function to send warning and errors to the Zope application log.

Here's an example of how to use the `LOG` function to write debugging messages:

```
      from zLOG import LOG, DEBUG
      LOG('my app', DEBUG, 'a debugging message')
```

You can use `LOG` in much the same way as you would use print statements to log debugging information while Zope is running. You should remember that Zope can be configured to ignore log messages below certain levels of severity. If you are not seeing your logging messages, make sure that Zope is configured to write them to the application log.

In general the debugger is a much more powerful way to locate problems than using the logger. However, for simple debugging tasks and for issuing warnings the logger works just fine.

Other Testing and Debugging Facilities

There is a few other testing and debugging techniques and tools not commonly used to test Zope. In this section we'll mention several of them.

Debug Logging

Zope provides an analysis tool for debugging log output. This output allows may give you hints as to where your application may be performing poorly, or not responding at all. For example, since writing Zope products lets your write unrestricted Python code, it's very possibly to get yourself in a situation where you "hang" a Zope request, possibly by getting into a infinite loop.

To try and detect at which point your application hangs, use the *requestprofiler.py* script in the *utilities* directory of your Zope installation. To use this script, you must run Zope with the `-M` command line option. This will turn on "detailed debug logging" that is necessary for the *requestprofiler.py* script to run. The *requestprofiler.py* script has quite a few options which you can learn about with the `--help` switch.

In general debug log analysis should be a last resort. Use it when Zope is hanging and normal debugging and profiling is not helping you solve your problem.

HTTP Benchmarking

HTTP load testing is notoriously inaccurate. However, it is useful to have a sense of how many requests your server can support. Zope does not come with any HTTP load testing tools, but there are many available. Apache's `ab` program is a widely used free tool that can load your server with HTTP requests.

Summary

Zope provides a number of different debugging and testing facilities. The debugger allows you to interactively test your applications. Unit tests allow help you make sure that your application is develops correctly. The logger allows you to do simple debugging and issue warnings.

To help maintain your sanity you should keeping your Zope products as simple as possible, use interfaces to describe functionality, and test your components outside as well as inside Zope.

Appendix A: Zope Core Permissions

This is a list of standard permissions included with Zope. It is a good idea to use these permissions when applicable with your Zope products, rather than creating new ones.

Core Permissions

Access contents information
: get "directory listing" info

Add Accelerated HTTP Cache Managers
: add HTTP Cache Manager objects

Add Database Methods
: add ZSQL Method objects

Add Documents, Images, and Files
: add DTML Method/Document objects, Image objects, and File objects

Add External Methods
: add External Method objects

Add Folders
: add Folder objects

Add MailHost objects
: add MailHost objects

Add Python Scripts
: Add Python Script objects

Add RAM Cache Managers
: Add RAM Cache manager objects

Add Site Roots
: add Site Root objects

Add User Folders
: add User Folder objects

Add Versions
: add Version objects

Add Virtual Host Monsters
: add Virtual Host Monster objects

Add Vocabularies
: add Vocabulary objects (ZCatalog−related)

Add ZCatalogs
: add ZCatalog objects

Add Zope Tutorials
: add Zope Tutorial objects

Change DTML Documents
: modify DTML Documents

Change DTML Methods
: modify DTML Methods

Change Database Connections
: change database connection objects

Change Database Methods
: change ZSQL method objects

Change External Methods
: change External Method objects

Change Images and Files
: change Image and File objects

Change Python Scripts
: change Python Script objects

Change Versions
: change Version objects

Change bindings
: change bindings (for Python Scripts)

Change cache managers
: change cache manager objects

Change cache settings
: change cache settings (cache mgr parameters)

Change configuration
: generic

Change permissions
: change permissions

Change proxy roles
: change proxy roles

Create class instances
: used for ZClass permission mappings

Delete objects
: delete objects

Edit Factories
: edit Factory objects (ZClass)

FTP access
: allow FTP access to this object

Import/Export objects
: export and import objects

Join/leave Versions
: join and leave Zope versions

Manage Access Rules
: manage access rule objects

Manage Vocabulary
: manage Vocabulary objects

Manage Z Classes
: Manage ZClass objects (in the control panel)

Manage ZCatalog Entries
: catalog and uncatalog objects

Manage properties
: manage properties of an object

Manage users
: manage Zope users

Open/Close Database Connections
: open and close database connections

Query Vocabulary
: query Vocabulary objects (ZCatalog−related)

Save/discard Version changes
: save or discard Zope version changes

Search ZCatalog
: search a ZCatalog instance

Take ownership

take ownership of an object

Test Database Connections
: test database connection objects

Undo changes
: undo changes to the ZODB (e.g. use the Undo tab)

Use Database Methods
: use ZSQL methods

Use Factories
: use Factory objects (ZClass−related)

Use mailhost services
: use MailHost object services

View
: view or execute an object

View History
: view ZODB history of an object

View management screens
: view management screens related to an object

Appendix B: Zope Directories

This is a list of some important directories in the Zope source code.

`Extensions`
: Code for External Methods go in this directory.

`ZServer`
: Python code for ZServer and Medusa.

`ZServer/medusa`
: Sam Rushing's Medusa package upon which ZServer is built.

`doc`
: Miscellaneous documentation.

`import`
: Place Zope export files here in order to import them into Zope.

`inst`
: Installation scripts.

`pcgi`
: C and Python code for PCGI.

`utilities`
: Miscellaneous utilities.

`var`
: Contains the ZODB data file (Data.fs) and various other files (logs, pids, etc.) This directory should be owned and writable by the userid that Zope is run as.

`lib/Components`
: Python extension modules written in C including BTree, ExtensionClass, cPickle, zlib, etc.

`lib/python`
: Most of the Zope Python code is in here.

`lib/python/AccessControl`
: Security classes.

`lib/python/App`
: Zope application classes. Stuff like product registration, and the control panel.

`lib/python/BTrees`
: Btrees package.

`lib/python/DateTime`
: DateTime package.

`lib/python/DocumentTemplate`
: DTML templating package. DTML Document and DTML Method use this.

`lib/python/HelpSys`
: Online help system.

`lib/python/Interface`
: Scarecrow interfaces package.

`lib/python/OFS`
: Object File System code. Includes basic Zope classes (Folder, DTML Document) and interfaces (ObjectManager, SimpleItem).

`lib/python/Products`
: Zope products are installed here.

`lib/python/Products/OFSP`
: The OFS product. Contains initialization code for basic Zope objects like Folder and DTML Document.

`lib/python/RestrictedPython`

Python security used by DTML and Python Scripts.

`lib/python/SearchIndex`
: Indexes used by ZCatalog.

`lib/python/Shared`
: Shared code for use by multiple Products.

`lib/python/StructuredText`
: Structured Text package.

`lib/python/TreeDisplay`
: Tree tag package.

`lib/python/ZClasses`
: ZClasses package.

`lib/python/ZLogger`
: Logging package.

`lib/python/ZODB`
: ZODB package.

`lib/python/ZPublisher`
: The Zope ORB.

`lib/python/Zope`
: The Zope package published by ZPublisher.

`lib/python/webDAV`
: WebDAV support classes and interfaces.