The Zope Book

Covers Zope 2.5

By Amos Latteier and Michel Pelletier

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Introduction

Welcome to *The Zope Book*. This book is designed to introduce you to Zope and its uses. Zope is an open−source web application server. If you are interested in writing web pages, programming web scripts, using databases, managing content, or doing a collaborative web development task, then you should read this book.

Why Should I Read this Book?

This book is meant to appeal to both current Zope users and people new to Zope:

- You don't need to be a programmer to read this book, or to use Zope.
- You should have some idea of how the web works; including a basic understanding of HTML and URLs.
- You should know what a web browser and a web server are and should have some idea of how they communicate.

The first part of the book explains to you how you use Zope through its web managment interface to manage dynamic content. The concepts in these chapters are are fundamental Zope concepts that show you how to use Zope to publish content on the web.

Some later sections of the book cover advanced topics such as relational databases, scripting with various programming languages, and XML. These chapters don't teach relational databases, programming, or XML, they simply show you how to use these technologies with Zope.

How the Book Is Organized

The organization of the book is presented below, as well as a brief summary of each chapter.

Part I: Introducing Zope

These chapters get the reader up and running with Zope. You learn about basic Zope objects and idioms.

Chapter 1: Introducing Zope

Chapter 1 explains what Zope is and who it's for. It describes in broad strokes what you can do with Zope. You also learn about the differences between Zope and other web application servers.

Chapter 2: Using Zope

Chapter 2 covers the most important Zope concepts. By the end of this chapter you should be able to use Zope to create and manage simple yet powerful web applications.

Chapter 3: Using Basic Zope Objects

Chapter 3 introduces *objects*, which are the most important elements of Zope. In it we cover what an object is in general, and then we introduce the basic Zope objects: folders, DTML documents, DTML methods, files, and images.

Chapter 4: Dynamic Content with DTML

Chapter 4 introduces *DTML*, Zope's tag−based scripting language. In it we describe DTML's use for templating and scripting and its place in relation to other ways to script Zope. We cover DTML syntax and the three most basic tags, *var*, *if* and *in*. After reading this chapter you'll be able to create dynamic web pages.

Chapter 5: Using Zope Page Templates

Chapter 5 introduces Zope Page Templates, a new tool to create dynamic HTML. This chapter shows you how to create and edit page templates. It also introduces basic template statements that let you insert dymanic content.

Chapter 6: Creating Basic Zope Applications

Chapter 6 walks the reader through several real−world examples of building a Zope application. It provides plenty of examples showing how to use Zope objects and how they can work together to form basic applications.

Part II: Creating Web Applications with Zope

These chapters provide a more in depth look at advanced Zope topics. They cover the material necessary to build real web applications with Zope.

Chapter 7: Users and Security

Chapter 7 looks at how Zope handles users, authentication, authorization, and other security−related matters. Security is central to Zope's design and should be central to the web applications that you create with Zope.

Chapter 8: Variables and Advanced DTML

Chapter 8 takes a closer look at DTML. It covers DTML security and the tricky issue of how variables are looked up in DTML. It also covers advanced uses of the basic tags covered in Chapter 3 and the myriad special purpose tags. This chapter will turn you into a DTML wizard.

Chapter 9: Advanced Page Templates

Chapter 9 goes into more depth with templates. This chapter teaches you all the template statements and expression types. It also covers macros which let you reuse presentation elements. By the end of this chapter you'll know all there is about page templates.

Chapter 10: Advanced Zope Scripting

Chapter 10 covers scripting Zope with Python and Perl. In it we cover how to write business logic in Zope using more powerful tools than DTML. It discusses the idea of *scripts* in Zope, and focuses on Python and Perl−based Scripts. This chapter shows you how to add industrial−strength scripting to your site.

Chapter 11: Searching and Categorizing Content

Chapter 11 shows you how to index and search objects with Zope's built−in search engine, the *Catalog*. It introduces indexing concepts and discusses different patterns for indexing and searching. Finally it discusses meta−data and search results. This chapter shows you how to create a powerful and easy to use information architecture.

Chapter 12: Relational Database Connectivity

Chapter 12 describes how Zope connects to external relational databases. It shows you how to connect to and query databases. It also covers features which allow you to treat relational data as though it were Zope objects. Finally, the chapter covers security and performance considerations.

Part III: Developing Advanced Web Applications with Zope

The final part of the book deals with advanced topics. You learn how to scale your web application and extend Zope itself.

Chapter 13: Scalability and ZEO

Chapter 13 covers issues and solutions for building and maintaining large web applications, and focuses on issues of management and scalability. In particular, the Zope Enterprise Option (ZEO) is covered in detail. This chapter shows you the tools and techniques you need to turn a small site into a large−scale site, servicing millions of visitors.

Chapter 14: Extending Zope

Chapter 14 covers extending Zope by creating your own classes of objects. It discusses *ZClasses*, and how instances are built from classes. It describes step by step how to build a ZClass and the attendant security and design issues. Finally, it discusses creating Python base classes for ZClasses and describes the base classes that ship with Zope. This chapter shows you how to take Zope to the next level, by tailoring Zope to your needs.

Conventions Used in This Book

This book uses the following typographical conventions:

Italic
: Italics indicate variables and is also used to introduce new terms.

`Fixed width`
: Fixed width text indicates commands, hyperlinks, and code listings.

Chapter 1: Introducing Zope

This chapter explains what Zope is and who it's for. It describes in broad strokes what you can do with Zope. You also learn about the differences between Zope and other web application servers.

What Is Zope?

Zope is a framework for building web applications. A web application is a computer program that users access with a web browser over the Internet. You can also think of a web application as a dynamic web site that provides not only static information to users but lets them use dynamic tools to work with an application.

Web applications are everywhere, and web users work with them all the time. Common examples of web applications are sites that let you search the web, like *Yahoo*, collaborate on projects, like *SourceForge*, or communicate with other people over e−mail, like *HotMail*. All of these kinds of applications can be developed with Zope.

So what do you get when you download Zope? You actually get a lot of things. Zope consists of several different components that work together to help you build web applications. Zope comes with:

A Web server
: Zope comes with a built in web server that serves content to you and your users. Of course, you may already have an existing web server, such as *Apache* or *Microsoft IIS* and you may not want to use Zope's. Not to worry, Zope works with these web servers also, and any other web server that supports the Common Gateway Interface (CGI).

A Web based interface
: When you build web applications with Zope, you use your web browser to interact with the Zope *management interface*. This interface is a development environment that lets you do things like create web pages, add images and documents, connect to external relational databases and write scripts in different languages.

An object database
: When you work with Zope, you are mostly working with objects that are stored in Zope's object database. Zope's management interface provides a simple, familiar way to manage objects that resembles the way many common file managers work.

Relational integration
: You don't have to store your information in Zope's object database if you don't want to, because Zope works with other relational databases such as *Oracle*, *PostgreSQL*, *Sybase*, *MySQL* and many others.

Scripting language support
: Zope allows you to write web applications in a number of different languages, like Python, Perl, and Zope's own Document Template Markup Language (DTML).

These are just some of the compelling features that have made Zope so popular for developing web applications. Perhaps Zope's best feature of all is its friendly, open source license. This means that not only is Zope free of cost for you to download, but you are also free to use Zope in your own products and applications without paying royalties or usage fees. Zope's open source license also means that all of the "source code" for Zope is available for you to look at, understand, and extend. Zope does not lock you into a proprietary solution that could hold you and your web users hostage.

From a business perspective, there are three key ideas to understanding what Zope can do for you: powerful collaboration, simple content management, and web components. The following sections are mostly oriented towards business people making decisions about Zope, so if you are interested in jumping right in, skip to the next chapter, *Using Zope*.

Powerful Collaboration

Years ago, Zope's core technologies were designed by Zope Corporation for an Internet Service Provider that provided web pages for newspapers. These newspapers in turn wanted to provide web pages for *their* customers. To scale in this environment, Zope was designed to safely delegate control to different groups of users at any level in the web site. Safely delegating control means considering these things:

1. Presenting information in an easy to understand way. Most people understand clicking on folders better than issuing database commands, so Zope uses an interface that resembles a simple file manager, like *Microsoft Windows Explorer* and other popular file managers.
2. Command line tools can be difficult to use and people are generally more comfortable using web browsers, so Zope was designed to be used almost exclusively through a web browser.
3. Collaborative environments require tools to allow users to recover from their mistakes and to work without interfering with each other, so Zope has Undo, Versions, and other tools to help people work safely together.

These features make Zope an ideal environment for programming and authoring web content by groups and sub−groups of users.

Simple Content Management

Many web applications are traditionally built in three layers. Data and other information is stored in databases, the programs that drive the behavior of the application are stored in files in one location, and the HTML and other layout and presentation information is stored somewhere else.

While this has advantages, it also has disadvantages. Different kinds of tools and expertise must be used to work with the different components. All the different components may need to have their own kind of security and maintenance concerns. Many of these kinds of tools are not manageable from a web browser or from simple command line or GUI tools like FTP.

In Zope, all of these components are brought together into one coherent system. All require a common set of services: security, web−based management, searching, clustering, syndication and others. By bringing together all of these concepts into one manageable system, Zope enables you to use one set of skills and one set of tools to develop complex web applications. In addition, centralizing our model means Zope can more easily work with other external tools, like relational databases, GUI web editors, and other systems that can inter−operate with Zope.

Web Components

The Web is a growing, dynamic platform. The web has evolved enough standards and APIs that creators of services, products, and technology can think in terms of the web as an architectural model to develop their applications around, instead of just a means of distributing static HTML documents to users.

Evidence of this is sprouting up in many locations. Microsoft's *.NET* architecture envisions a world of web components running on remote systems, providing specific services to applications around the world. *Frontier*, by UserLand Software, pioneered a simple web services protocol called XML−RPC to allow web components to communicate with each other (Zope also works with XML−RPC, which is discussed in Chapter 10, "Advanced Zope Scripting"). With web components, the model of a person sitting in front of the browser is no longer the only model of the web.

Zope History

In 1996 Jim Fulton, the CTO of Zope Corporation and Python guru, was drafted to teach a class on CGI programming, despite not knowing much about the subject. Jim studied all of the existing documentation on CGI on his way to the class. On the way back from the class, Jim considered what he didn't like about traditional CGI based programming environments: its fragility, lack of object−orientation, and how it exposes web server details. From these initial musings, the core of Zope was written on the plane flight back from the class.

Zope Corporation went on to release three open source software packages to support web publishing, *Bobo*, *Document Template*, and *BoboPOS*. These package were written in Python. They have evolved into core components of Zope providing the web ORB (Object Request Broker), DTML scripting language, and object database. Zope is still mostly written in Python with a few performance−critical sections in C.

Back then, Zope Corporation had developed a commercial application server based on their three open source components. This product was called *Principia*. In Novermber of 1998, investor Hadar Pedhazur convinced Zope Corporation to open source Principia. This became Zope, which was given its own home at Zope.org.

Who Can Benefit from Zope?

It takes a lot of people working together to create web services. To manage and coordinate these people on large−scale sites can be a difficult task. We've identified some common roles in this scenario to be aware of:

- *Consumers* use the site to locate and work with useful content.
- *Business Users* create and manage the site's content.
- *Site Designers* create the site's look and feel.
- *Site Developers* program the site's services.
- *Component Developers* create software intended for distribution.
- *Administrators* keep the software and environment running.
- *Information Architects* make platform decisions and keep track of the big picture.

Zope is a platform upon which Site Developers create services to be turned over to Site Designers and Business Users, and Component Developers distribute new products and services for Zope users world wide.

Zope can install Zope products that are focused on different audiences. For instance, *Squishdot* is a popular weblog, written in Zope, that is useful right out of the box. Squishdot users won't necessarily see that Zope is underneath. Other Zope products, such as Zope Corporation's Content Management Framework, take the same approach, targeting audiences that need not know of Zope's existence underneath.

How Can You Benefit From Zope?

We've looked at the Zope philosophy and architecture, now let's survey some of of the applications of Zope. All sites solve different problems, but many sites tackle a set of common issues daily. Here are some of the main uses of Zope:

To Present Dynamic Content
: You want to tailor your web site's presentation to its users, integrate information in databases and provide users with searching. You'd also like to make your web site automate and facilitate your business processes. Can your web site react intelligently to visitors in order to provide a compelling experience? Zope allows you to make every page dynamic. It comes with facilities for personalization, integrating information in databases and searching.

To Manage your Web Site
: A small web site is easy to manage, but a web site that serves thousands of documents, images and files needs to provide powerful management tools. Can you manage your site's data, business logic and presentation all in one place? Can you keep up with your content, or is it getting out of hand? Zope gives you simple and powerful tools for handling gigabytes of web content. You can manage your logic, presentation and data all from your web browser.

To Secure Your Web Site
: When you deal with more than a handful of web users, security becomes very important. It is crucial to organize users and be able to safely delegate tasks to them. For example, folks in your engineering department may need to be able to manage their web pages and business logic, designers may need to update site templates, and database administrators need to manage database queries. Can your system handle thousands of users, perhaps linked to your existing LDAP or other user databases with flexible security rules? Zope allows you to scale your site to thousands of site managers and millions of visitors. You can simply control security policies and safely delegate control to others.

To Provide Network Services
: Right now most web sites serve users, but soon web sites will need to serve remote computer programs and other web sites. For example, you'd like to make your news items automatically available to wire service web sites. Or maybe you want to make products for sale on your site automatically searchable from a product comparison site. Can you leverage your existing data and business logic to create network services or will you have to start over from scratch? Zope's built−in support for networking makes every Zope site a network service. Your business logic and data can be accessed over the web via HTTP and XML−RPC.

To Integrate Diverse Content
: Your content is strewn all over the place, in relational databases, files, web sites, FTP archives, XML. Can you unify your data into one coherent application? Does your system support web standards so that you can integrate content from legacy systems and new systems that you will add in the future? Zope supports web standards allowing you to use your existing data, infrastructure and filesystems.

To Provide Scalability
: So you struck it rich and now you're getting more hits than you ever imagined. Now you need to handle a dramatically greater level of traffic than before. Can you move your site to a different database and server platform and spread the load across multiple servers? Can your web site grow to handle your success? Zope allows your web applications to scale across as many machines as necessary to handle your load. Zope makes it possible to maintain a small site that can turn into a huge site overnight based on its "ZEO" technology (see Chapter 13, "Scalability and ZEO" for more details).

What Zope Gives You

Let's take a closer look at the Zope features that allow you to build and manage dynamic web sites.

Unique Management Environment
: The first thing you'll notice about Zope is that it lets you manage your site's data, logic, and presentation right in your web browser. This means that Zope is easy to use and is remotely administrable. Zope lets you collaborate with others to interactively develop your web site.

Built−in Tools
: Zope comes with site management tools, a web server, a search engine, database connectivity, security and collaboration services, and more. Out of the box, Zope gives you everything you need to build a powerful web site.

Open Standards Support
: Zope excels at gluing together diverse data because of its support for open standards. Zope supports Internet standards including SQL, ODBC, XML, DOM, FTP, HTTP, FastCGI, XML−RPC, SOAP, and more.

Open Source Licensing
: With Zope you don't just get an application, you get the source and a community. Since Zope is open source you are not held hostage by a single vendor; you're free to use, distribute and adapt Zope to fit your needs. Zope also benefits from an active user and developer community. The community improves Zope's support, audits Zope's security, fixes bugs, and adds features.

Extensibility
: Zope can be extended in many directions. Third party applications can be easily created and distributed. The Zope community has produced hundreds of Zope add−ons for everything from credit card processing to web discussions.

Zope Alternatives

There are many tools available to help you build web applications. Early in the history of the web, simple web applications were built almost exclusively with CGI programs written in Perl or other languages. Now there are a host of options ranging from open source frameworks like PHP, to commercial options such as Allaire's Cold Fusion, Java application servers, and Vignette's Story Server.

Zope offers a unique mix of features, some similar to, and some very different from, features offered by other web application tools. Zope is easy to use, open source, powerful, and provides support for many different kinds of applications. Here is a short list of common web tool drawbacks and Zope's advantages:

- Some tools do not offer a simple file manager like user interface, and are hard to use. Zope has a simple user interface.
- Some tools require complex configuration. Zope is easy to install and requires no configuration before you begin using it.
- Some tools require using unfamiliar and proprietary development tools. Zope works with any standards−compliant web browser and no other tools will be required to use this book.
- Some tools don't scale as well as Zope does to handle large numbers of developers and users. Zope has a consistent, powerful user management system that can scale to many users with unique, easily managed privileges.
- Finally most closed−source, commercial tools don't let you extend, customize, and redistribute them. Zope is open source.

Zope Community

Zope was one of the first tools of its kind to become open source. Since opening up the code to Zope, the user base has grown tremendously.

The Zope community consists of Zope users and developers. Many of the community members are professionals such as consultants, developers and web masters, investing their time and money into supporting Zope. Others are students and curious hackers, learning how to use a cool new tool. The community gets together occasionally at conferences but spends most of its time discussing Zope on the many Zope mailing lists and web sites. You can find out more about the many Zope−related mailing lists at http://www.zope.org/Resources/MailingLists.

Now that you've learned about Zope's features and history, it's time to start using it. In the next chapter you'll learn how to get up and running with Zope. Since Zope is free, you can download the latest version, and begin working immediately.

Chapter 2: Using Zope

This chapter gets you up and running with Zope. It guides you through installing and running Zope. This chapter covers the most important Zope concepts. By the end of this chapter you should be able to use Zope to create and manage simple yet powerful web applications.

Downloading Zope

The first steps to using Zope are to download and install it. Zope is available for free from the Zope.org web site. The most recent stable version is always available from the Download section of Zope.org.

Zope is currently available as a binary for Windows, Linux and Solaris. This means that you can just download and install it without having to compile any programs. For other platforms you must download the source and compile Zope. Zope can be compiled and run on almost any Unix−like operating system. As a general rule of thumb, if Python is available for your operating system and you have a C compiler, then you can probably use Zope.

Installing Zope

You will install Zope differently depending on your platform. If you are running a recent version of Linux, you may already have Zope installed. You can get Zope in both binary and source forms. There are also several different binary formats available.

Installing Zope for Windows

Zope for Windows comes as a self−installing *.exe* file. To install Zope, double click on the installer. The installer walks you through the installation process. Pick a name for your Zope installation and a directory to install it in. Click *Next* and create a new Zope user account. This account is called the *initial user*. This creates an account that you can use to log into Zope for the first time. You can change this user name and password later if you wish.

If you are using Windows NT or Windows 2000, you can choose to run Zope as a service. Running Zope as a service is a good idea for a public server. If you are just running Zope for personal use don't bother running it as a service. Keep in mind that if you are running Windows 95, Windows 98, or Windows ME (Millenium Edition), you cannot run Zope as a service.

If you decide to uninstall Zope later you can use the *Unwise.exe* program in your Zope directory.

Downloading Linux and Solaris Binaries

Download the binary for your platform and extract the tarball:

```
        $ tar xvfz Zope−2.4.0−linux2−x86.tgz
```

In this example, you are downloading version 2.4.0. This may not be the most recent version of Zope when you read this, so be sure and get the latest *stable* version of Zope for your platform.

This will unpack Zope into a new directory. Enter the Zope directory and run the Zope installer script:

```
        $ cd Zope−2.4.0−linux2−x86
        $ ./install
```

The installer will print information as it installs Zope. Among other things, it will create a initial user account. You can change the initial user name and password later with the *zpasswd.py* script (see Chapter 7, "Users and Security").

The installer will configure Zope to run as your UNIX userid. If you prefer to run Zope as another userid, you can use the `-u` command line switch and specify the user you want to configure Zope to run as. There are many books out there with more information on userids and UNIX administration in general you should check out if you want to do anything fancy. For now things will work fine if you just install Zope to runs as your userid by not specifying any extra command line options.

For more information on installing Zope see the installation instructions in *doc/INSTALL.txt* and find out more about the installer script by running it with the `-h` help switch:

```
        $ ./install -h
```

Getting Zope in *RPM* and *deb* format

Zope Corporation doesn't make Zope available in RPM format, but other people do. Jeff Rush regularly packages Zope as RPMs. For more information check out his web page (http://www.taupro.com/Downloads/Zope/). Zope is also available in the Debian Linux *deb* package format. You can find Zope deb packages at the Debian web site site. Generally the latest Zope releases are found in the *unstable* distribution.

Compiling Zope from Source Code

If binaries aren't available for your platform, then chances are you can compile Zope from the source. To do this, install Python from the sources for your platform and make sure you have a C compiler. You can get Python from the Python.org web site. Although we try and use the most recent Python for Zope, often the latest Python version is more recent than the version we "officially" support for Zope. For information on which version of Python you need to compile Zope with, see the release notes on the Web page for each version. Zope 2.4 requires Python 2.1. Zope 2.3 and earlier versions used Python 1.5.2.

Download the Zope source distribution and extract the tarball:

```
        $ tar xvfz Zope-2.4.0-src.tgz
```

This unpacks Zope into a new directory. Enter the Zope directory and run the Zope installer script:

```
        $ cd Zope-2.4.0-src
        $ python wo_pcgi.py
```

The installer compiles Zope and sets up your installation. The installer prints information as it runs, including the initial user name and password. It's important to *write down* that information so you can log into Zope. For more information see the installation instructions in the file *doc/INSTALL.txt*. You can change the initial user account later with the *zpasswd.py* script (see Chapter 7, "Users and Security").

Starting Zope

Depending on your platform, you run Zope with different commands. Whatever your platform, you can either run Zope manually, or automatically. When running Zope manually, you simply tell Zope when to start and when to stop. When running Zope automatically, Zope will start and stop when your computer starts and stops.

Starting Zope On Windows

The installer creates a Zope directory with a batch file called *start.bat*. Double click the *start.bat* icon. This will open a window that includes logging information. On this window you find out what port Zope is listening on. You can now log into Zope with a web browser.

If you are running Zope as a service, you can start and stop Zope via the Services control panel. Zope will write events to the event log so that you can keep track of when your service starts and stops. If you run Zope as a service you must know what port Zope is running on, since you will not have direct access to its detailed logging information.

Zope comes with its own web server. When you start Zope, its web server starts. If you wish you can connect Zope to your existing web server such as IIS, but this is beyond the scope of this book. The Zope Administrator's Guide covers this kind of material.

Starting Zope on UNIX

Run the *start* script:

```
$ ./start &
```

Zope will start running and will print logging information to the console. You should see information telling you what port Zope is listening on. You can now log into Zope with a web browser.

Zope comes with its own web server. When you start Zope, its web server starts. If you wish you can connect Zope to your existing web server such as Apache, but this is beyond the scope of this book. The Zope Administrator's Guide covers this kind of material.

The *start* script can also be edited to start Zope with many different options. How to customize your Zope startup is also described in the Administrator's Guide.

Logging In

To log into Zope you need a web browser. Zope's interface is written entirely in HTML, therefore any browser that understands modern HTML works. Mozilla, and any 3.0+ version of Microsoft Internet Explorer or Netscape Navigator will do.

To log into the management interface point your web browser to Zope's management URL. The management URL for Zope is Zope's base URL with */manage* appended. Assuming you have Zope installed on your local machine serving on the default port 8080, the management URL is:

```
http://localhost:8080/manage
```

This URL usually works, but you may need to login to a different machine than the one we show you here. To find out exactly which URL to use, look at the logging information Zope prints as it starts up. For example:

```
------
2000-08-07T23:00:53 INFO(0) ZServer Medusa (V1.18) started at Mon Aug  7 16:00:53 2000
        Hostname: peanut
        Port:8080

------
2000-08-07T23:00:53 INFO(0) ZServer FTP server started at Mon Aug  7 16:00:53 2000
```

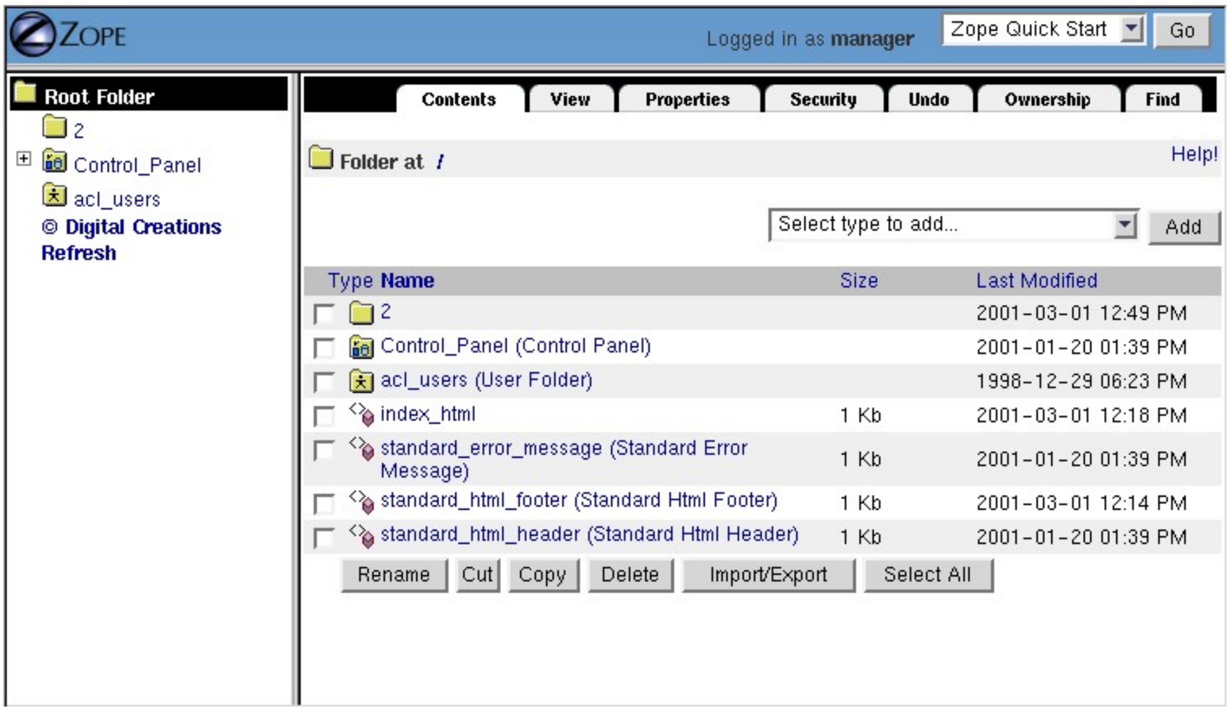
```
                Authorizer:None
                Hostname: peanut
                Port: 8021
------
2000−08−07T23:00:53 INFO(0) ZServer Monitor Server (V1.9) started on port 8099
```

The first log entry indicates Zope is running on a machine named *peanut* and that the web server is listening on port 8080. This means that the management URL is *http://peanut:8080/manage*. Later in the book we'll look at the other servers referred to in the logging information.

After you enter Zope's management URL in your browser, your browser will prompt you to log into Zope by providing a user name and password. Type in the initial user name and password created during the install process. If you don't know the initial user name and password, then shut Zope down by closing its window, and change the initial user password with the *zpasswd.py* script and restart Zope. See Chapter 7, "Users and Security" for more information about configuring the initial user account.

Controlling Zope with Management Interface

After you successfully login you see a web page of the Zope management interface, as shown in Figure 2−1.



Type Name	Size	Last Modified
2		2001-03-01 12:49 PM
Control_Panel (Control Panel)		2001-01-20 01:39 PM
acl_users (User Folder)		1998-12-29 06:23 PM
index_html	1 Kb	2001-03-01 12:18 PM
standard_error_message (Standard Error Message)	1 Kb	2001-01-20 01:39 PM
standard_html_footer (Standard Html Footer)	1 Kb	2001-03-01 12:14 PM
standard_html_header (Standard Html Header)	1 Kb	2001-01-20 01:39 PM

Figure 2−1 The Zope management interface.

The Zope management interface lets you control Zope within your web browser.

Using the Navigator

The Zope management interface is broken into three frames. With the left frame you navigate around Zope much like you would navigate around a file system with a file manager like Windows Explorer. This frame is called the *Navigator*, and is shown in the left frame of Figure 2−1. In this frame you see the root folder and all of its subfolders. The root folder is in the upper left corner of the tree. The root folder is the "top" of Zope. Everything in Zope lives inside the root folder.

Some of the folders have plus marks to the left of them. These plus marks let you expand the folders to see the sub−folders that are inside.

Above the folder tree Zope shows you login information in a frame. In this screen shot you can see that you are currently logged in as "manager". When you log in to Zope you will use the initial user account and you will see the name of this account in place of "manager".

To manage a folder, click on it and it will appear in the right−hand frame of the browser window. This frame is called the *workspace*.

Using The Workspace

The right−hand frame of the management interface shows the object you are currently managing. When you first log into Zope the current object is the root folder, as shown in the right frame of Figure 2−1. The workspace gives you information about the current object, and lets you change it.

Across the top of the screen are a number of tabs. The currently selected tab is highlighted in a lighter color. Each tab takes you to a different *view* of the current object. Each view lets you perform a different management function on that object.

In Figure 2−1, you are looking at the *Contents* view of the root folder object.

At the top of the workspace, just below the tabs, is a description of the current object's type and URL. On the left is an icon representing the current object's type, and to the right of that is the object's URL.

In Figure 2−1 "Folder at /" tells you that the current object is a folder and that its URL is `/`. Note that this URL is the object's URL relative to Zope's base URL. So if the URL or your Zope site was *http://mysite.example.com:8080*, then the URL of the "Folder at /myFolder" would be *http://mysite.example.com:8080/myFolder*.

As you explore different Zope objects, you find that the URLs (as displayed in the management screen), can be used to navigate between objects.

For example, if you are managing a folder at */Zoo/Reptiles/Snakes* you can return to the folder at */Zoo* by clicking on the word *Zoo* in the folder's URL.

In the frame at the top of the management interface, your current login name is displayed, along with a pull−down box that lets you select:

Preferences
: Here, you can set default preferences for your Zope session, you can even set to hide the top frame.

Logout
: Selecting this menu item will log you out of Zope.

Quick Start Links
: These are quick links out to Zope documentation and community resources.

Understanding Users in Zope

Zope is a multi−user system. You've already seen how you can log into Zope via the management interface with the initial user name and password. Zope supports other kinds of users:

Emergency User

The emergency user is rarely used in Zope. This account is used for creating other user accounts and fixing things if you accidently lock yourself out.
The emergency user is both very powerful and very weak. It is not restricted by most security controls. However, the emergency user can only create one type of object: Users. Using the Emergency User to repair your Zope system in the case of accidental lock−out is discussed in the Administrator's Guide.

Manager

The Manager is the Zope workhorse. You will need to log in with the Manager account to do most of the work involved with building Zope web sites. The initial user is a Manager, and you can create as many Manager accounts as you need.

Others

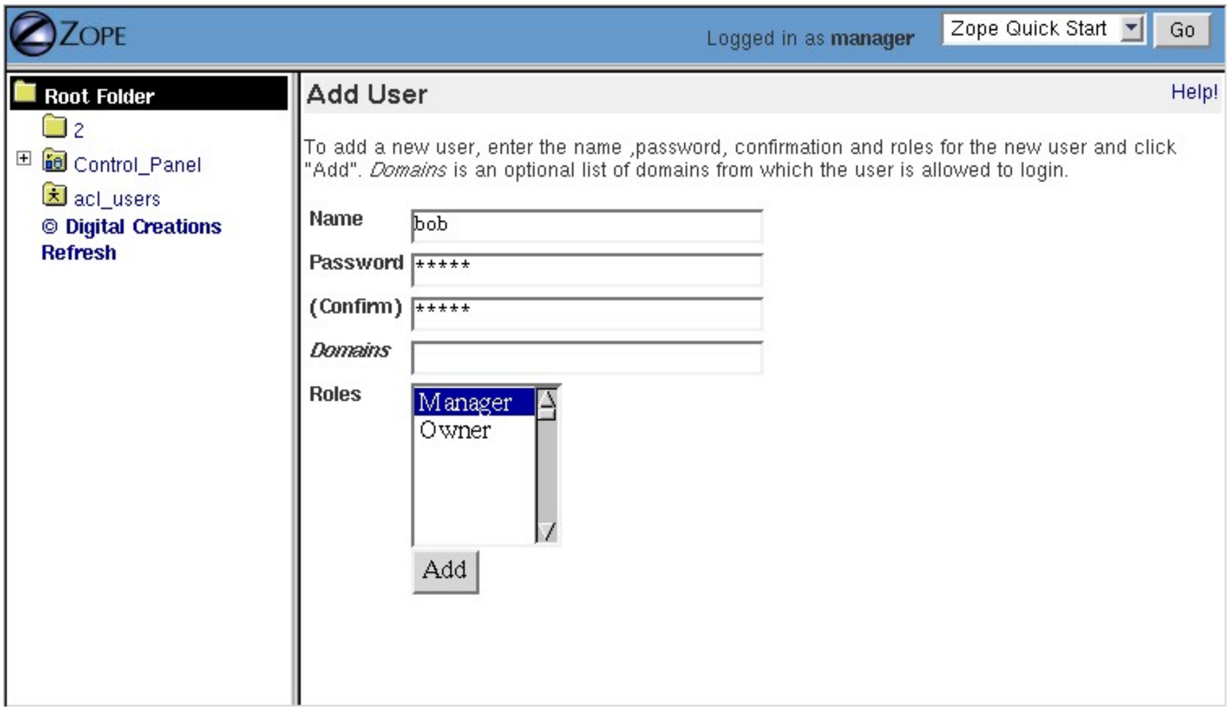
You can create your own kind of users that fit into groups, or are responsible for carrying out a role that you define. This is explained more in Chapter 7, "Users and Security", which discusses Zope security and users.

Creating Users

Managers can create Zope users in a unique kind of folder called a *User Folder*.

User folder icons look like folders with a person on them. User folders always have the name *acl_users*, as shown in Figure 2−1.

Click on the acl_users folder in the root folder to enter it. User folders contain User objects. You can create new users and edit existing users. Click the *Add* button to create a new user, as shown in Figure 2−2.



Add User

To add a new user, enter the name ,password, confirmation and roles for the new user and click "Add". *Domains* is an optional list of domains from which the user is allowed to login.

Name	bob
Password	*****
(Confirm)	*****
Domains	
Roles	Manager (selected), Owner

Figure 2−2 Adding a new user.

Fill out the form to create a new user. In the *Name* field put your chosen user name. Choose a password and enter it in the *Password* and *(Confirm)* fields. Leave the *Domains* field blank. This an advanced feature and is discussed in Chapter 7, "Users and Security". Select the *Manager* role from the *Roles* select list. Then click the *Add* button.

Congratulations, you've just created a manager account. Zope will show you this new manager account inside the user folder. Later you can change or delete this user if you wish.

Changing Logins

To change your login select *Logout* from the top frame of the management interface. You will be prompted to login again. To change logins, enter a new user name and password.

To logout select *Logout* from the top frame of the management interface and cancel the new login. You should see a message telling you that you are logged out. If you try to access the Zope management interface after you are logged out, you'll be prompted to log in again. You can also logout of Zope by quitting your web browser.

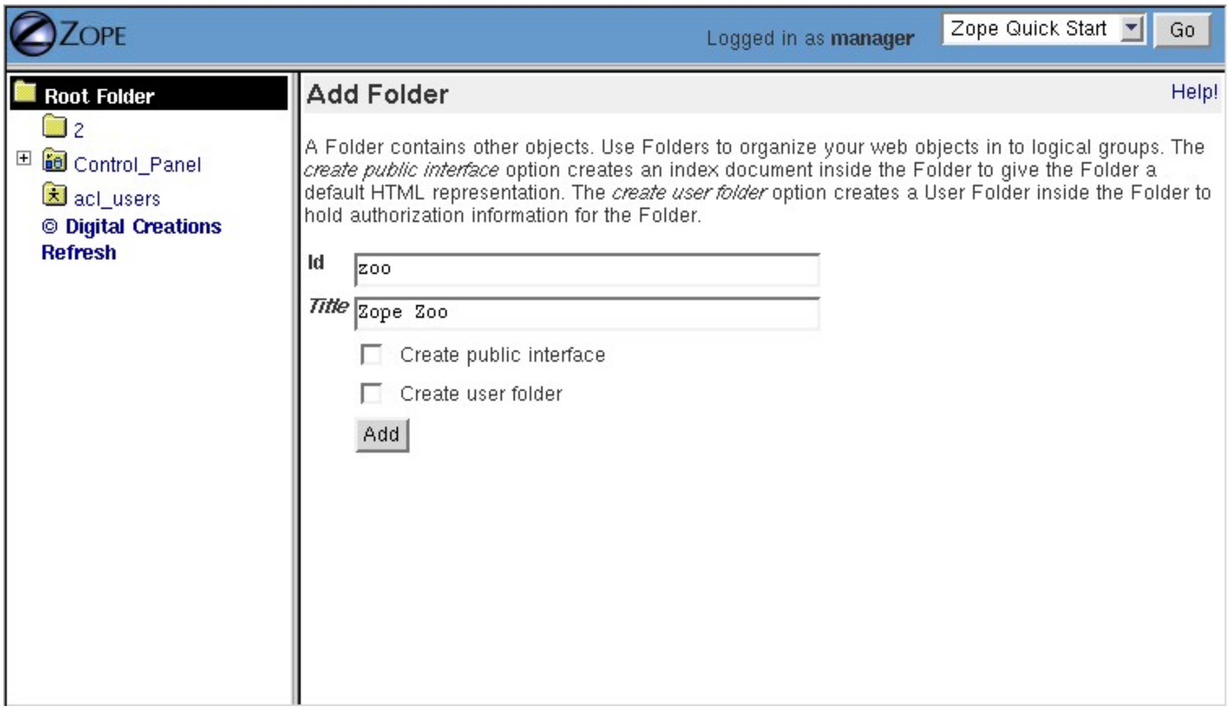
Creating Objects

The Zope management interface represents everything in terms of objects and folders. When you build web applications with Zope, you spend most of your time creating and managing objects in folders. For example, to make a new manager account you create a user object in a user folder.

Return to the root folder by clicking on the top left folder in the Navigator frame.

To add a new object to the current folder select an object from the pull−down menu in the right frame labeled "Select type to add...". This pull−down menu is called the *product add list*.

For example, to create a folder, select *Folder* from the product add list. At this point, you'll be taken to an add form that collects information about the new folder, as shown in Figure 2−3.



A Folder contains other objects. Use Folders to organize your web objects in to logical groups. The *create public interface* option creates an index document inside the Folder to give the Folder a default HTML representation. The *create user folder* option creates a User Folder inside the Folder to hold authorization information for the Folder.

Id: zoo

Title: Zope Zoo

☐ Create public interface

☐ Create user folder

Add

Figure 2−3 Folder add form.

Type "zoo" in the *Id* field, and "Zope Zoo" in the *Title* field. Then click the *Add* button.

Zope will create a new folder in the current folder. You can verify this by noting that there is now a new folder named *zoo* inside the root folder.

Click on *zoo* to enter it. Notice that the URL of the folder is based on the folder's id. You can create more folders inside your new folder if you wish. For example, create a folder inside the *zoo* folder with an id of *arctic*. Go to the *zoo* folder and choose *Folder* from the pull−down menu. Then type in "arctic" for the folder id, and "Arctic Exhibit" for the title. Now click the *Add* button. You always create new objects in the same way:

1. Go to the folder where you want to add a new object.
2. Choose the type of object you want to add from the pull−down menu.
3. Fill out an add form and submit it.
4. Zope will create a new object in the current folder.

Notice that every Zope object has an id that you need to specify in the add form when you create the object. The id is how Zope names objects. Objects also use their ids for their URLs.

Chapter 3, "Using Basic Zope Objects", covers all of the basic Zope objects and what they can do for you.

Moving Objects

Most computer systems let you move files around in directories with cut, copy and paste. The Zope management interface has a similar system that lets you move objects around in folders by cutting or copying them, and then pasting them to a new location.

To experiment with copy and paste, create a new folder in the root folder with an id of *bears*. Then select *bears* by checking the check box just to the left of the folder. Then click the *Cut* button. Cut removes the selected objects from the folder and places them on a clipboard. The object will *not*, however, disappear from its location until it is pasted somewhere else.

Now enter the *zoo* folder by clicking on it, and then enter the *arctic* folder by clicking on it. You could also have used the Navigator to get to the same place. Now, click the *Paste* button to paste cut object(s) into the current folder. You should see the *bears* folder appear in its new location. You can verify that the folder has been moved by going to the root folder and noting that *bears* is no longer there.

Copy works similarly to cut. When you paste copied objects, the original objects are not changed. Select the object(s) you want to copy and click the *Copy* button. Then navigate to another folder and click the *Paste* button.

You can cut and copy folders that contain other objects and move many objects at one time with a single cut and paste. For example, go to the *zoo* folder and copy the *arctic* folder. Now paste it into the *zoo* folder. You will now have two folders inside the *zoo* folder, *arctic* and *copy of arctic*. If you paste an object into the same folder where you copied it, Zope will change the id of the pasted object. This is a necessary step, as you cannot have two objects with the same id in the same folder.

To rename the *copy of arctic* folder, select the folder by checking the check box to the left of the folder. Then click the *Rename* button. This will take you to the rename form as shown in Figure 2−4.

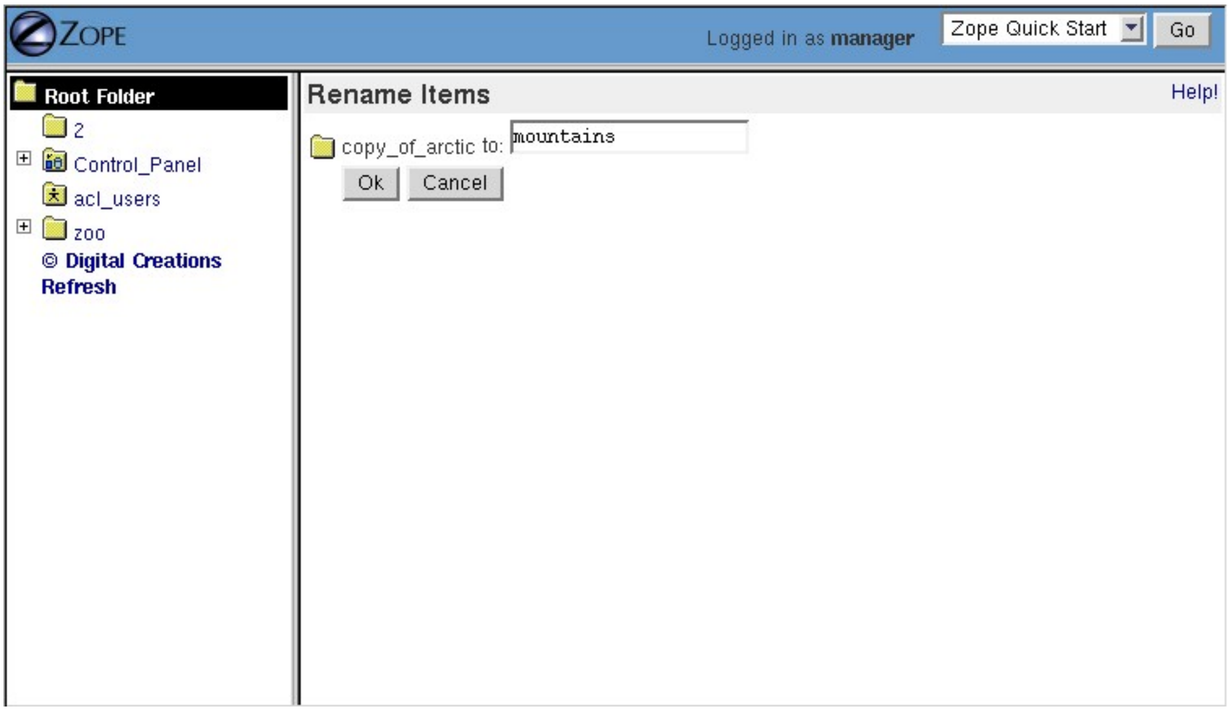


Figure 2−4 Renaming an Object

Type in a new id "mountains" and click *OK*. Zope ids can consist of letters, numbers, spaces, dashes underscores and periods, and are case−sensitive. Here are some legal Zope ids: *index.html*, *42*, and *Snake−Pit*.

Now your *zoo* folder contains an *arctic* and a *mountains* folder. Each of these folders contains a *bears* folder. This is because when we made a copy of the *arctic* folder it also copied the *bears* folder that it contained.

If you want to delete an object, select it and then click the *Delete* button. Unlike cut objects, deleted objects are not placed on the clipboard and cannot be pasted. In the next section we'll see how we can retrieve deleted objects.

Zope will not let you cut, delete, or rename a few particular objects in the root folder. These objects include *Control_Panel*, *standard_html_header*, *standard_html_footer*, and *standard_error_message*. These important objects are necessary for Zope's operation. Also, these operations don't work in some cases. For instance, you can't paste a user object into a regular folder.

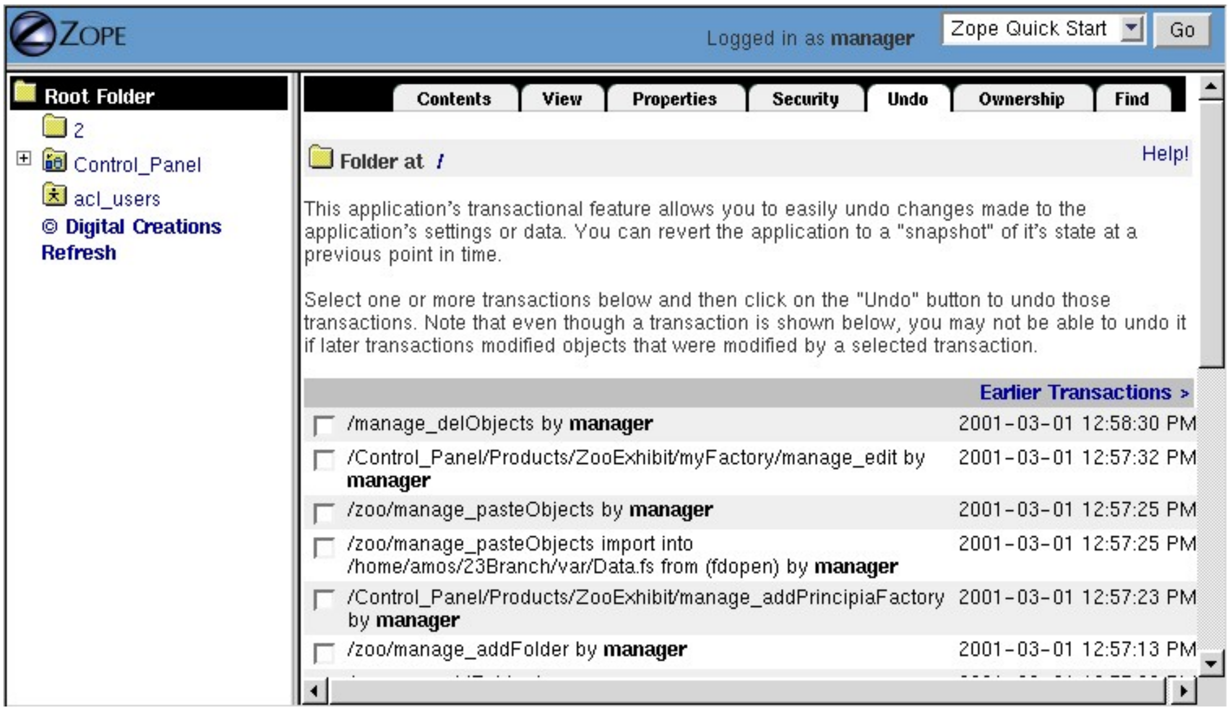
If you are having problems with copy and paste, make sure that you have enabled cookies in your browser. Zope uses cookies to keep track of the objects that you cut and copy.

Undoing Mistakes

Any action in Zope that causes objects to change can be undone, via the *Undo* tab. You can recover from mistakes by undoing them.

Select the *zoo* folder that we created earlier and click *Delete*. The folder disappears. You can get it back by undoing the delete action.

Click the *Undo* tab, as shown in Figure 2−5.



This application's transactional feature allows you to easily undo changes made to the application's settings or data. You can revert the application to a "snapshot" of it's state at a previous point in time.

Select one or more transactions below and then click on the "Undo" button to undo those transactions. Note that even though a transaction is shown below, you may not be able to undo it if later transactions modified objects that were modified by a selected transaction.

Earlier Transactions >

/manage_delObjects by **manager**	2001-03-01 12:58:30 PM
/Control_Panel/Products/ZooExhibit/myFactory/manage_edit by **manager**	2001-03-01 12:57:32 PM
/zoo/manage_pasteObjects by **manager**	2001-03-01 12:57:25 PM
/zoo/manage_pasteObjects import into /home/amos/23Branch/var/Data.fs from (fdopen) by **manager**	2001-03-01 12:57:25 PM
/Control_Panel/Products/ZooExhibit/manage_addPrincipiaFactory by **manager**	2001-03-01 12:57:23 PM
/zoo/manage_addFolder by **manager**	2001-03-01 12:57:13 PM

Figure 2−5 The Undo view.

Select the first transaction labeled */manage_delObjects*, and click the *Undo* button.

This action tells Zope to undo the last transaction. You can verify that the task has been completed by making sure that the *Zoo* folder has returned.

Undo Details and Gotchas

Undo works on the object database that Zope uses to store all Zope objects. Changes to the object database happen in transactions. You can think of a transaction as any change you make to Zope, such as creating a folder or pasting a bunch of objects to a new place. Each transaction describes all of the changes that happen in the course of performing the action.

You cannot undo a transaction that a later transaction depends upon. For example, if you paste an object into a folder and then delete an object in the same folder you might wonder whether or not you can undo the earlier paste. Both transactions change the same folder so you can not simply undo the earlier transaction. The solution is to undo both transactions. You can undo more than one transaction at a time by selecting multiple transactions on the *Undo* tab and then clicking *Undo*.

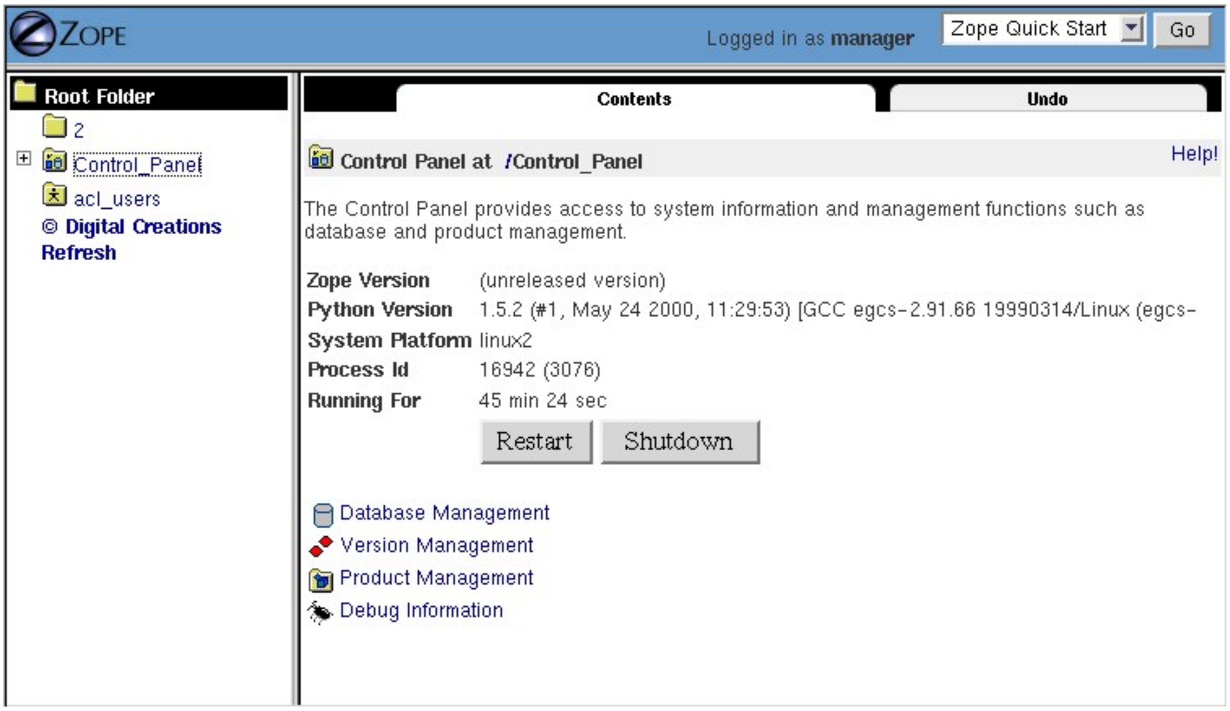
Another problem to be aware of is that you cannot undo an undo. Therefore if you add a folder and then undo that particular action, you cannot get the new folder back by undoing the undo.

One last note on undo. Only changes to objects stored in Zope's object database can be undone. If you have integrated data in a relational database server such as Oracle or MySQL (as discussed in Chapter 12, "Relational Database Connectivity") changes to data stored there cannot be undone.

Administering and Monitoring Zope

The Control Panel is an object in the root folder that controls various aspects of Zope's operation.

Click on the *Control_Panel* object in the root folder, as shown in Figure 2−6.



Control Panel at /Control_Panel

The Control Panel provides access to system information and management functions such as database and product management.

```
Zope Version     (unreleased version)
Python Version   1.5.2 (#1, May 24 2000, 11:29:53) [GCC egcs-2.91.66 19990314/Linux (egcs-
System Platform  linux2
Process Id       16942 (3076)
Running For      45 min 24 sec
```

Figure 2−6 The Control Panel

To shutdown Zope, click the *Shutdown* button. Shutting down Zope will cause the server to stop handling requests and completely exit from memory. You will have to manually start Zope to continue using it. Only shutdown Zope if you are finished using it, and have the ability to access the server on which Zope is running, so that you can manually restart it later.

If you are running Zope on UNIX under daemon control or as a service on Windows, you can restart Zope from the control panel folder. Clicking the *Restart* button will shut down Zope and then immediately start up a new instance of the Zope server. It may take Zope a few seconds to come back up and start handling requests.

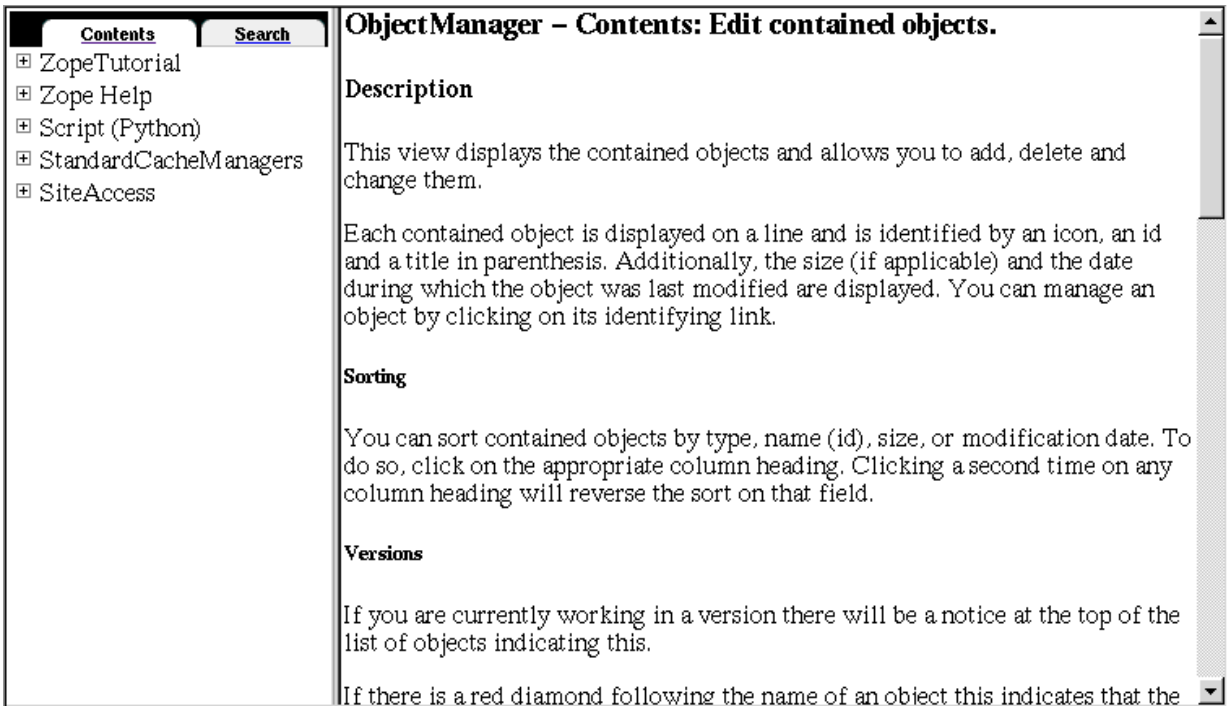
On the control panel you will also see several links at the bottom of the screen, one of which is *Database Managment*, which controls Zope's object database.

Transactions don't go away until you pack the Zope database. This means that you can undo all transactions except ones that have been removed by packing the database. When you choose to pack the database you can specify which transactions to remove so that you can for example only remove transactions older than a week.

Using the Help System

Zope has a built in help system. Every management screen has a help button in the upper right−hand corner. This button launches another browser window and takes you to the Zope Help System.

Go to the root folder. Click the Help button, and you should see what is shown in Figure 2−7.



ObjectManager – Contents: Edit contained objects.

Description

This view displays the contained objects and allows you to add, delete and change them.

Each contained object is displayed on a line and is identified by an icon, an id and a title in parenthesis. Additionally, the size (if applicable) and the date during which the object was last modified are displayed. You can manage an object by clicking on its identifying link.

Sorting

You can sort contained objects by type, name (id), size, or modification date. To do so, click on the appropriate column heading. Clicking a second time on any column heading will reverse the sort on that field.

Versions

If you are currently working in a version there will be a notice at the top of the list of objects indicating this.

If there is a red diamond following the name of an object this indicates that the

Figure 2−7 The Help System.

The help system has an organization similiar to the two primary panes of the Zope management interface, it has one frame for navigation and one for displaying the current topic.

Whenever you click the help button from the Zope management screen, the right frame of the help system displays the help topic for the current management screen. In this case, you see information about the *Contents* view of a folder.

Browsing and Searching Help

Normally you use the help system to get help on a specific topic. However, you can browse through all of the help content if you are curious, or simply want to find out about things besides the management screen you are currently viewing.

The help system lets you browse all of the help topics in the *Contents* tab of the left−hand help frame, as shown in Figure 2−7. You can expand and collapse help topics. To view a help topic in the right frame, click on it.

All help on the Zope management screens is located in the *Zope help* folder. Inside you'll find many help topics. You'll also find a help folder called *API Reference*. This folder contains help on scripting Zope, which is explained further in Chapter 9, "Advanced Zope Scripting".

When you install third−party components they also include help that you can browse. Each installed component has its own help folder.

Search the help system by clicking on the Search tab and entering one or more search terms. For example, to find all of the help topics that mention folders, type "folder" into the search form.

Starting with the Zope Tutorial

Zope comes with a built−in tutorial. The tutorial guides you through all the basics of creating and managing Zope objects. To launch the tutorial, add a Zope Tutorial to the current folder by selecting *Zope Tutorial* from the Product add list. Give it an id which is unique in the current folder, such as *tutorial*. The tutorial comes with several examples that you can change and copy for your own use.

If you start the tutorial and want to stop using it before you have completed all the lessons, you can later return to the tutorial. Just go to the help system and find the lesson you'd like to continue with by browsing the *Zope Tutorial* help folder. There is no need to re−install the tutorial.

If you are having problems with the tutorial, make sure to enable cookies in your browser. The tutorial uses cookies to keep track of where your example objects are. Also, if you enable Javascript in your browser, the tutorial will make sure that the Zope management interface stays in sync with your tutorial lesson.

Now that you have Zope running, it's time to explore the system more thoroughly. You've seen how to manage Zope through the web and have learned a little about Zope objects. In the next chapter you'll meet many different Zope objects and find out how to build simple applications with them.

Chapter 3: Using Basic Zope Objects

When building a web application with Zope, you construct the application out of objects. By design, different objects handle different parts of your application. Some objects hold your content data, such as word processor documents, spreadsheets and images. Some objects handle your application's logic by accepting input from a web form, or executing a script. Some objects control the way your content is displayed, or *presented* to your viewer, for example, as a web page, or via email. In general Zope objects take three types of roles:

Content
: Zope objects such as documents, images and files hold different kinds of textual and binary data. In addition to objects in Zope containing content, Zope can work with content stored externally, for example, in a relational database.

Logic
: Zope has facilities for scripting business logic. Zope allows you to script behavior using Python, Perl, and SQL. "Business logic" is any kind of programming that does not involve presentation, but rather involves carrying out tasks like changing objects, sending messages, testing conditions and responding to events.

Presentation
: You can control the look and feel of your site with Zope objects that act as web page templates. Zope comes with a tag based scripting language called the Document Template Markup Language (DTML) to control presentation.

The word *object* is a heavily loaded term. Depending on your background, it may mean any number of different things. In this chapter, you can think of a Zope object as an application component that you can control and edit using a web browser.

Zope comes with many built−in objects that help you perform different tasks. You can also install third party Zope objects to expand Zope's capabilities. This chapter explains the most basic objects and how they work. You can create fully functional Zope sites using the few basic objects that are covered in this chapter.

This chapter is loosely structured around the above three categories, Content, Logic, and Presentation. There are other kinds of objects in Zope that don't clearly fit into one of these three roles; those are explained at the end of the chapter.

Using Zope Folders

Folders are the building blocks of Zope. The purpose of a folder is to *contain* other objects, and to *organize* objects by separating them into different groups.

Folders can contain all sorts of objects, including other folders, so you can nest folders inside each other to form a tree of folders. This kind of folder within a folder arrangement provides your Zope site with *structure*. Good structure is very important, as almost all aspects of Zope (from security to behavior to presentation) are influenced by your site's folder structure.

Folder structure should be very familiar to anyone who has worked with files and folders on their computer with a file manager program like Microsoft *Windows Explorer* or any one of the popular UNIX file managers like *xfm*, *kfm*, or the Gnome file manager. The Zope management interface tries to look as much as possible like these popular programs so that you are familiar with how to organize Zope objects just like you would organize files on your computer.

Managing Folder Contents

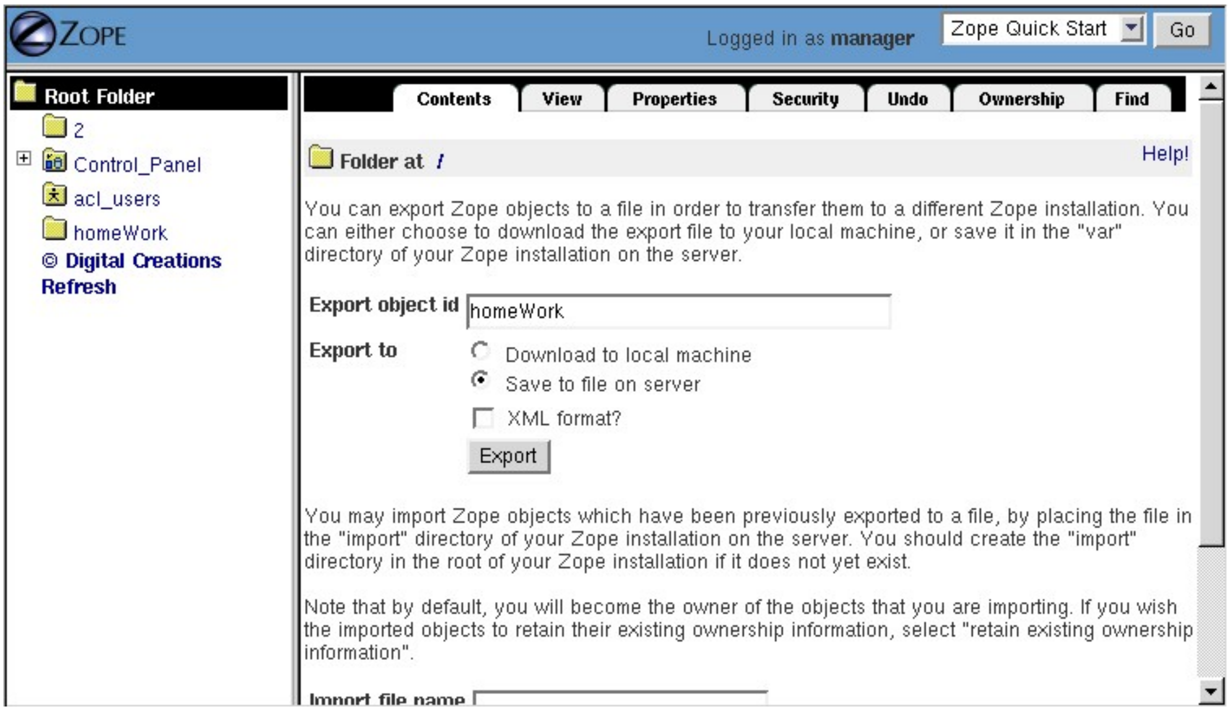
In Chapter 2, "Using Zope" you created objects and moved objects around. In summary, you create objects in folders by choosing the type of the object you are looking for from the pull−down menu at the top of the folder's *Contents* view. Then you fill out the add form and submit it. A new object is then added to the current folder. You can move objects between folders using the *Cut*, *Copy*, *Paste*, *Delete*, and *Rename* buttons.

Importing and Exporting Objects

You can move objects from one Zope system to another using *export* and *import*. You can export all types of Zope objects to an *export file*. This file can then be imported into any other Zope system.

You can think of exporting an object as cloning a piece of your Zope system into a file that you can then move around from machine to machine. You can take this file and graft the clone onto any other Zope server. Imagine you had some documents in a Zope folder. If you wanted to copy just those objects to your friend's Zope system, you could export the folder and send the export file via email to the friend, who could then import it.

Suppose you have a folder for home work that you want to export from your school Zope server, and take home with you to work on in your home Zope server. You can create a folder like this in your root folder called "homeWork". Go to the folder that contains your *homeWork* folder. Select the *homeWork* folder by checking the checkbox next to it. Then click the *Import/Export* button. You should now be working in the Import/Export folder view, as shown in Figure 3−1.



You can export Zope objects to a file in order to transfer them to a different Zope installation. You can either choose to download the export file to your local machine, or save it in the "var" directory of your Zope installation on the server.

Export object id: homeWork

Export to: ( ) Download to local machine [✓ Save to file on server]

[ ] XML format?

Export

You may import Zope objects which have been previously exported to a file, by placing the file in the "import" directory of your Zope installation on the server. You should create the "import" directory in the root of your Zope installation if it does not yet exist.

Note that by default, you will become the owner of the objects that you are importing. If you wish the imported objects to retain their existing ownership information, select "retain existing ownership information".

Import file name

Figure 3−1 The Import/Export View

There are two sections to this screen. The upper half is the export section and the lower half is the import section. To export an object from this screen, type the id of the object into the first form element, Export object id. In our case Zope already filled this field in for us, since we selected the *homeWork* folder on the last screen.

The next form option lets you choose between downloading the export file to your computer or leaving it on the server. If you check *Download to local machine*, and click the Export button, your web browser will prompt you to download the export file. If you check *Save to file on server*, then Zope will save the file on the same machine on which Zope is running, and you must fetch the file from that location yourself. The export file will be written to Zope's *var* directory on your server. By default export files have the *.zexp* file extension.

In general it's handier to download the export file to your local machine. Sometimes it's more convenient to save the file to the server instead, for example if you are on a slow link and the export file is very large, or if you are just trying to move the exported object to another Zope instance on the same computer.

The final export form element is the *XML format?* checkbox. Checking this box exports the object in the *eXtensible Markup Language* (XML) format. Leaving this box unchecked exports the file in Zope's binary format. The XML format is much bigger to download, but is human readable and XML parsable. For now, the only tool that understands this XML format is Zope itself, but in the future there may be other tools that can understand Zope's XML format. In general you should leave this box unchecked unless you're curious about what the XML format looks like and want to examine it by hand.

Click the Export button and save your *homeWork.zexp* export file.

Now suppose that you've gone home and want to import the file into your home Zope server. First, you must copy the export file into Zope's *import* directory on your server. Now, go to the *Import/Export* view of the folder where you want perform the import. Enter the name of the export file in the *Import file name* form element and click *Import* to import those objects into Zope.

Zope gives you the option to either *Take ownership of imported objects* or *Retain existing ownership information*. Ownership will be discussed more in Chapter 7, "Users and Security". For now, just leave the *Take ownership of imported objects* option selected.

After you import the file you should have a new object in the Zope folder where you performed the import.

To bring your homework back to school, perform the same export and import procedure. Note that you cannot import an object into a folder that has an existing object with the same id. Therefore, when you bring your homework back to school, you'll need to import it into a folder that doesn't already have a *homeWork* folder in it. Then, you'll need to delete your old *homeWork* folder and copy the newly imported one into its place.

Temporary Folders

Temporary Folders are Zope folders that are used for storing temporary objects. Temporary Folders acts almost exactly like a regular Folder with three significant differences:

1. Everything contained in a Temporary Folder disappears when you restart Zope.
2. You cannot undo actions taken to objects stored a Temporary Folder.
3. You cannot use a Version to manipulate objects in a Temporary Folder

By default there is a Temporary Folder in your root folder named *temp_folder*. You may notice that there is an object entitled, "Session Data Container" within *temp_folder*. This is an object used by Zope's default sessioning system configuration. See the "Using Sessions" section later in this chapter for more information about sessions.

Temporary folders store their contents in RAM rather than in the Zope database. This makes them appropriate for storing small objects that receive lots of writes, such as session data. However, it's a bad idea use temporary folders to store large objects because your computer can potentially run out of RAM as a result.

Using Zope Page Templates

In Zope 2.5, a new, powerful type of object was added called *Page Templates*. Page templates allow you to define dynamic presentation for a web page by writing an HTML template. The HTML in your template is made dynamic by inserting special XML namespace elements to your HTML which define the dynamic behavior for that page.

Page templates are powerful for a few reasons:

- They are always valid HTML. There is no need to insert invalid code into your templates like you would with other dynamic languages.
- Page templates separate logic from presentation. By intentionally only focusing on the goals of presentation, page templates do not allow you to use them as a general purpose programming language.
- HTML designers do not need to be programmers. Because page templates start from an HTML `mock up` your HTML designers do not need to know anything about programming to develop the initial design of a page template.
- Your programmers do not need to be HTML designers. Page templates also have a benefit the other direction, because your programmers can make your templates dynamic by just adding XML tag attributes, they can experiment with different dynamic behaviors without destroying or rewriting your original HTML.

Creating Zope Page Templates

Create a Folder called *Sales* in the root folder. Click on the Sales folder and then select *Page Template* from the add list you learned about in Chapter 2. This process will take you to the add form for a page template. Specify the id "SalesPage" and the title "Template for Sales Staff" and click *Add*. You have successfully created a page template. However, it's content is standard boilerplate text, so move on to the next step to edit the content.

Editing Zope Page Templates

The easiest way to edit a page template is by clicking on its name or icon in the Zope management interface. When you click on either one of those items, you are taken to the *Edit* view of the page template which gives you a text−area where you can edit the template. Replace the original content that comes with the page template with the following HTML:

```
<html>
  <body>
    <h1>This is my first page template!</h1>
  </body>
</html>
```

and click *Save*. Now you can click on the *View* tab to view the page template. This particular template does not do anything special or have any dynamic behavior. In later sections in this chapter, we'll add some dynamic behavior. In Chapter 5, you'll use page templates in much greater detail to create dynamic presentation.

Uploading Zope Page Templates

Suppose you'd prefer not to edit your HTML templates in a web browser, or you have some existing HTML pages that you'd like to bring into Zope. Zope allows you to upload your existing html files and convert them to page templates.

Select *Page Template* from the add menu, this will take you to the add form for page templates that we saw earlier.. The last form element on the add form is the *Browse* button. Click this button. Your browser will then pop up a file selection dialog box. Select the text file on your computer that you want to upload to this template.

Type in an *Id* for the new Document and click *Add*. After clicking *Add*, you will be taken back to the management screen. There you will see your new page template.

Using Zope Documents

Documents hold text. In web applications, you generally use documents to create web pages. You can also use documents to hold text files or snippets of text or HTML code such as sidebars or headers. In addition to containing text, a document allows you to edit the text through the web. Zope has several different types of documents. The most important is DTML Document. DTML stands for *Document Template Markup Language*.

There are other third−party object types (generally called "Products") available from sources such as Zope.org which will extend your installation to support other types of textual and non−textual content.

DTML Documents

Use DTML Documents to create web pages and sections of documents, such as a sidebars that can be shared by web pages. DTML Documents can contain scripting commands in DTML (Zope's tag based scripting language). The mix of HTML and DTML generates dynamic web pages.

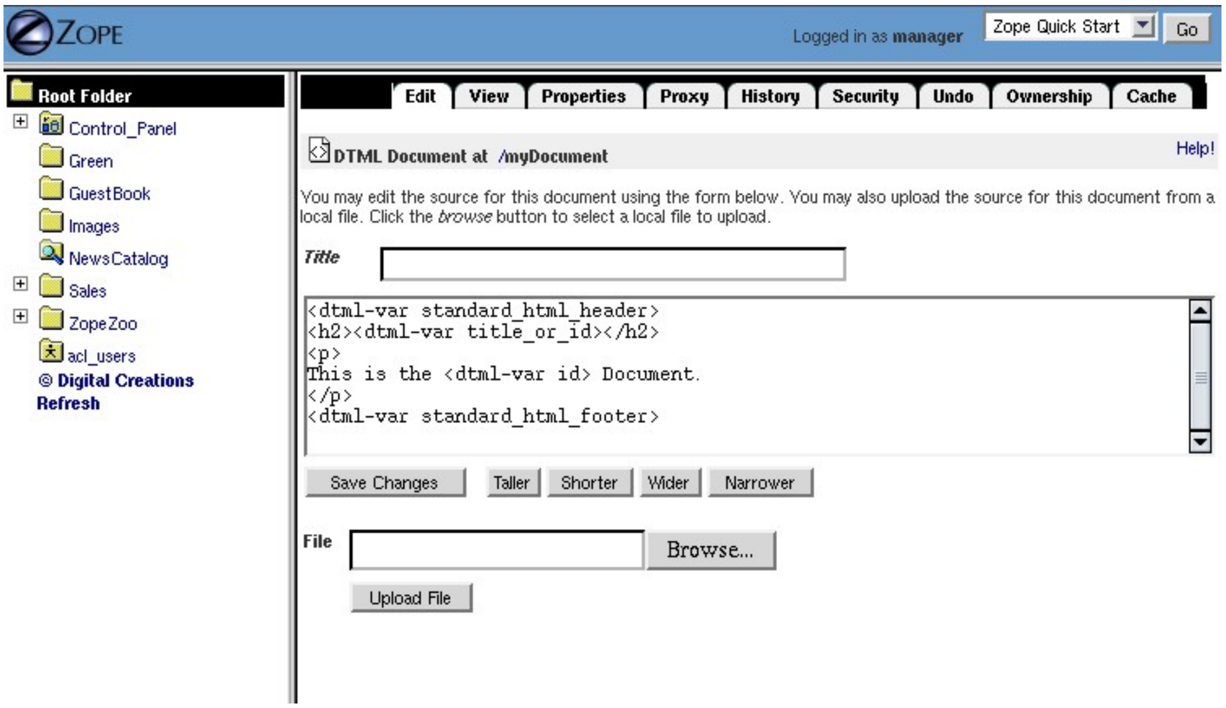
DTML Documents are also useful for creating shared content, such as common document structures.

Creating DTML Documents

Click on the Sales folder and then select *DTML Document* from the add list. This process will take you to the add form for a DTML Document. Specify the id "SalesStaff" and the title "The Jungle Sales Staff" and click *Add*. You have successfully created a DTML Document. However, its content is standard boilerplate text, so move on to the next step to edit the content.

Editing DTML Documents

The easiest and quickest way to edit a DTML Document is through the management interface. To select a document, click on its name or icon, which will bring up the form shown in Figure 3−2.



DTML Document at /myDocument

You may edit the source for this document using the form below. You may also upload the source for this document from a local file. Click the *browse* button to select a local file to upload.

```
<dtml-var standard_html_header>
<h2><dtml-var title_or_id></h2>
<p>
This is the <dtml-var id> Document.
</p>
<dtml-var standard_html_footer>
```

Figure 3−2 Editing a DTML Document

This view shows a text area in which you can edit the content of your document. If you click the *Change* button you make effective any changes you have made in the text area. You can control the size of the text area with the *Taller*, *Shorter*, *Wider*, and *Narrower* buttons. You can also upload a new file into the document with a the *File* text box and the *Upload File* button.

Delete the default content that is automatically inside the current *SalesStaff* DTML Document.

Add the following HTML content to the *SalesStaff* document:

```
<html>
<body>
<h2>Jungle Sales Staff</h2>

<ul>
  <li>Tarzan</li>
  <li>Cheetah</li>
  <li>Jane</li>
</ul>
</body>
</html>
```

After you have completed the changes to your document, click the *Change* button. Zope returns with a message telling you that your changes have taken effect. Now, you can look at the document by clicking the *View* tab.

Congratulations! You've just used Zope to create an HTML page. You can carry out the creation and editing in one step rather than two by using the *Add and Edit* button on the add page.

You can edit your HTML online and view it immediately. In fact, you can create entire Zope sites of HTML documents and folders. This process shows just the surface of Zope's benefits, but it provides a good way to familiarize yourself with Zope. You can also write some dynamic content in Zope and let those who are interested purely in design edit their own HTML web pages in this way.

Uploading an HTML File

Suppose you'd prefer not to edit your HTML files in a web browser, or you have some existing HTML pages that you'd like to bring into Zope. Zope allows you to upload your existing text files and convert them to DTML Documents.

Select *DTML Document* from the add menu, this will take you to the add form for DTML Documents. The last form element on the add form is the *Browse* button. Click this button. Your browser will then pop up a file selection dialog box. Select the text file on your computer that you want to upload to this document.

Type in an *Id* for the new Document and click *Add*. After clicking *Add*, you will be taken back to the management screen. There you will see your new document.

Viewing DTML Documents

The primary purpose of a DTML document is to hold useful content. This content's primary usage is to be viewed. DTML Documents can be viewed several different ways:

Management Interface
: From the management interface you can Click on a Document's *View* tab to view the contents of the document.

Calling Directly Through the Web
: Documents can be called directly through the web by going to their URL location with a web browser.

Called by Another Object
: Other objects, especially other DTML objects, can display a Document's contents.

Calling Through the Web

Like all Zope objects, a DTML Document's URL is based on its id. For example, if you have a DTML Document in the root folder called *Bob*, then its URL would be:

```
http://localhost:8080/Bob
```

If *Bob* is in a sub−folder called *Uncles* then its URL would be:

```
http://localhost:8080/Uncles/Bob
```

There could also be other DTML Documents in the Uncles folder called *Rick*, *Danny* and *Louis*. You access them through the web similarly:

```
http://localhost:8080/Uncles/Rick
http://localhost:8080/Uncles/Danny
http://localhost:8080/Uncles/Louis
```

Translating URLs to objects isn't a new idea, web servers like Apache do it all the time. They translate URLs to files and directories on a filesystem. Zope carries this simple idea to greater heights. In Zope, URLs are always simple to read because they map easily and simply onto the way objects are organized in Zope. This is why we told you that your site's structure is key to your site's success.

Going directly to the URL of a DTML Document is called *calling it through the web*. This causes the content of the DTML Document to be evaluated and returned to your web browser. In the next chapter on DTML, we will see what it means for DTML to be evaluated, but for now, you can easily experiment with DTML and simple HTML content to get the idea.

Calling from Another Object

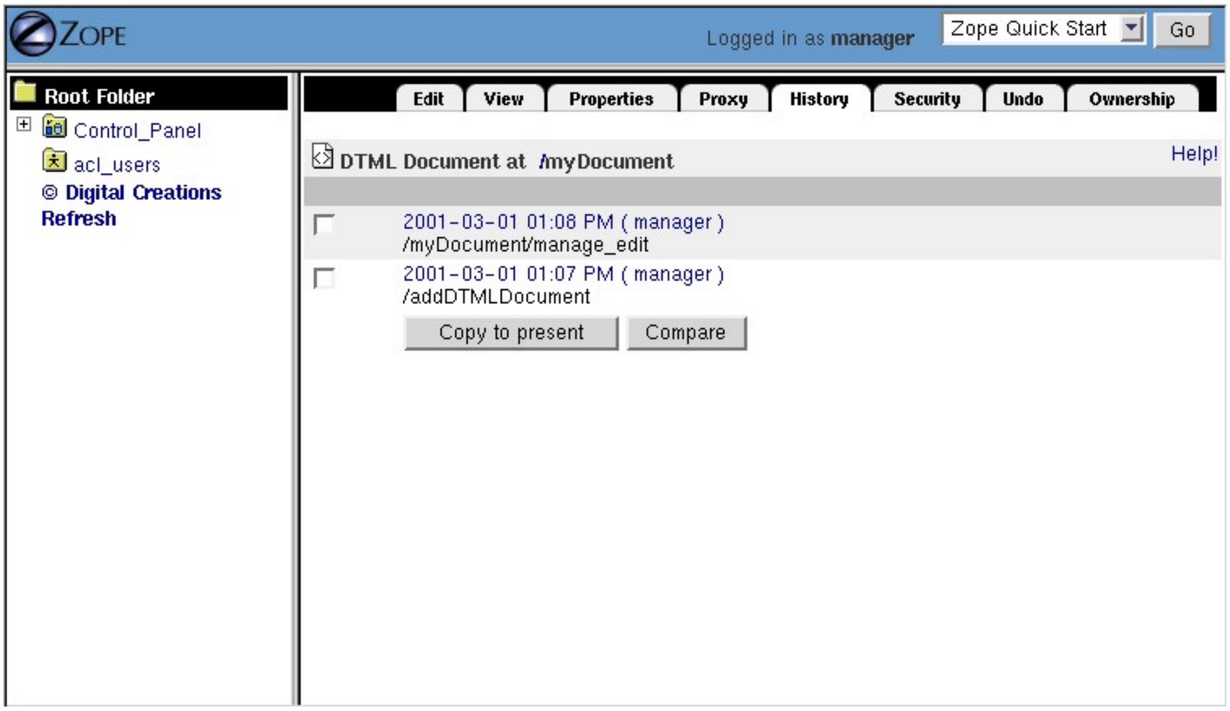
In using Zope you probably have encountered examples of DTML like this:

```
<dtml-var standard_html_header>

  <h1>This is some simple HTML</h1>

<dtml-var standard_html_footer>
```

Here we see that one DTML object, *standard_html_header* is being called by the document that contains this code. In this case, the evaluated contents of the first document are *inserted* into the contents of this calling document. This is a very fundamental concept in Zope and will be used throughout the book.

Reviewing Changes to Documents

The Undo tab lets you undo one transaction at a time, but often it is useful to undo only the change to one object. Remember, a transaction can be a group of actions all taken at the same time. If a document was edited in a transaction that also included moving an object, you may just want to undo the change to the document, but *not* undo moving the file. To do that, you can go to that object's *History* View and look at the previous states of the object, as shown in Figure 3−4.



DTML Document at /myDocument

2001-03-01 01:08 PM (manager)
/myDocument/manage_edit

2001-03-01 01:07 PM (manager)
/addDTMLDocument

Figure 3−4 The History View

Documents even support the idea of comparing revisions, allowing you to track changes to your objects. For example, DTML Methods and Documents will allow you to select two revisions and compare them to one another. You many want to use this to see what people have done to your object, for example, let's say you had a document that contained a list of all the animals in a Zoo. If one of your co−workers then goes and edits that list and saves it, you can use the history comparison feature to compare the most recent "new" version of the file with the next most recent version.

This comparison is displayed in a popular format called *diff*. The diff shows you the lines that have been added to the new document (via a plus), which lines have been subtracted from the old document (via a minus), and which lines have been replaced or changed (via an exclamation point).

Remote Editing with FTP, WebDAV, and PUT

Zope lets you edit documents directly in your web browser, though this is not the only way documents can be edited in Zope. For simple documents, editing through the web is a handy method. But for large, complex documents, or documents that have special formatting, it's useful to be able to use the editor you are most used to.

DTML Documents can be edited with FTP, WebDAV, and the HTTP PUT protocol. Many HTML and text editors support these protocols for editing documents on remote servers. Each of these protocols has advantages and disadvantages:

FTP
: FTP is the File Transfer Protocol. FTP is used to transfer a file from one computer to another. Many text editors support FTP, so it is very useful.

WebDAV
: WebDAV is a new Internet protocol based on the Web's underlying protocol, HTTP. DAV stands for Distributed Authoring and Versioning. Because DAV is new, it may not be supported by as many text editors as FTP.

PUT
: The HTTP protocol supports a simple way to upload content to a server called PUT. PUT is supported by many HTML editors, such as Netscape Composer.

Using one of these methods, you can edit your content with a variety of tools. In the next couple sections, we'll show you a couple simple tools that use FTP to edit Zope content.

Uploading Documents and Files with WS_FTP

WS_FTP is a popular FTP client for Windows that you can use to upload documents and files into Zope with the FTP protocol. WS_FTP can be downloaded from the Ipswitch Home Page.

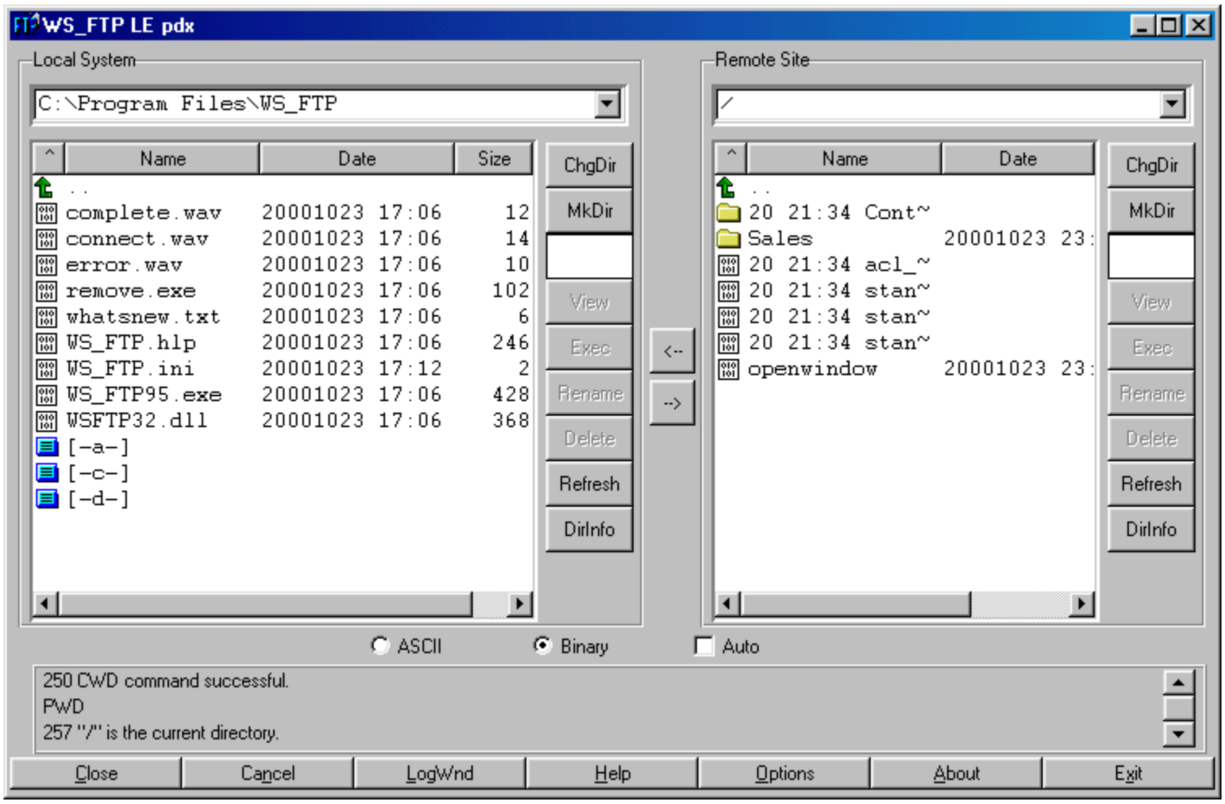
There are other popular Windows FTP clients, and many web browsers like Netscape and Microsoft Internet Explorer come with FTP clients also. This section applies to other FTP clients also.

In Chapter 2, "Using Zope" you determined the URL of your Zope system by looking at the start up log. Finding out how to contact your Zope's FTP server follows a similar process:

```
        ------
        2000-08-07T23:00:53 INFO(0) ZServer Medusa (V1.18) started at Mon Aug  7
  16:00:53 2000
                Hostname: peanut
                Port:8080

        ------
        2000-08-07T23:00:53 INFO(0) ZServer FTP server started at Mon Aug  7
  16:00:53 2000
                Authorizer:None
                Hostname: peanut
                Port: 8021
        ------
        2000-08-07T23:00:53 INFO(0) ZServer Monitor Server (V1.9) started on port
  8099
```

The startup log says that the Zope FTP server is listening to port 8021 on the machine named *peanut*. When you start WS_FTP, you will need to know the machine name and port information so you can connect to Zope via FTP. After typing in the machine name and port of your Zope server, hit the Connect button. WS_FTP will now ask you for a username and password. Enter your management username and password for the Zope management interface.

If you type in your username and password correctly, WS_FTP shows you what your Zope site looks like through FTP. There are folders and documents that correspond exactly to what your root Zope folder looks like through the web, as shown in Figure 3−3.



```
C:\Program Files\WS_FTP
..
complete.wav   20001023 17:06    12
connect.wav    20001023 17:06    14
error.wav      20001023 17:06    10
remove.exe     20001023 17:06   102
whatsnew.txt   20001023 17:06     6
WS_FTP.hlp     20001023 17:06   246
WS_FTP.ini     20001023 17:12     2
WS_FTP95.exe   20001023 17:06   428
WSFTP32.dll    20001023 17:06   368
[-a-]
[-c-]
[-d-]
```

```
/
..
20 21:34 Cont~
Sales          20001023 23:
20 21:34 acl_~
20 21:34 stan~
20 21:34 stan~
20 21:34 stan~
openwindow     20001023 23:
```

```
250 CWD command successful.
PWD
257 "/" is the current directory.
```

Figure 3−3 Editing Zope through FTP

Transferring files to and from Zope is a very easy task with WS_FTP. On the left−hand side of the WS_FTP window is a file selection box that represents files on your local machine. The file selection box on the right−hand side of the WS_FTP window represents objects in your Zope system. Transferring files from your computer to Zope or back again is as easy as selecting the file you want to transfer and clicking either the left arrow (download) or the right arrow (upload). WS_FTP has lots of cool features and customizations that you can use to make remote object management with Zope very easy.

Editing Zope Objects with Emacs

Emacs is a very popular text editor. In fact, Emacs is more than just a text editor, it is a whole culture. Emacs comes in two flavors, GNU Emacs and XEmacs. Both of these flavors of Emacs can work directly over FTP to manipulate Zope documents and other textual content.

Emacs will let you treat any remote FTP system like any other local filesystem, making remote management of Zope content an easy process. Therefore, you need not leave Emacs in order to use Zope.

Emacs provides a richer set of text editing capabilities than most web browser text areas. Emacs can be used to directly edit documents and manipulate objects through FTP, therefore Emacs is a nice Zope development environment.

By default when you start up Zope, Zope runs an FTP server just as it runs an HTTP server. You can specify when you start Zope which port the FTP server should listen on, but by default this port is 8021.

To log into Zope, run Emacs. The file you visit to open an FTP connection depends on which text editor you are running: XEmacs or Emacs:

Xemacs

To visit a remote file in XEmacs, visit a file by the form:

```
/user@server#port:/
```

This will open a connection to the / folder of the FTP server running on *server* and listening on port *port*.

Emacs

To visit a remote file in Emacs, visit a file by the form:

```
/user@server port:/
```

The literal space is inserted by holding down the Control key and the Q key, and then pressing the space "C−Q ".

For the typical Zope installation with XEmacs, the filename to open up an FTP session with Zope is */user@localhost#8021:/*.

Emacs will ask you for a password to log into Zope's FTP server.

Visiting the / folder of an FTP server in Zope, Emacs will list the contents of the root folder:

```
      drwxrwx---   1 Zope     Zope            0 Dec 30  1998 Control_Panel
      drwxrwx---   1 Zope     Zope            0 Dec 30  1998 QuickStart
      drwxrwx---   1 Zope     Zope            0 Dec 30  1998 Sales
      -rw-rw----   1 Zope     Zope         1024 May  3  1999 index_html
      -rw-rw----   1 Zope     Zope         1381 May  3  1999
standard_error_message
      -rw-rw----   1 Zope     Zope           55 Dec 30  1998
standard_html_footer
      -rw-rw----   1 Zope     Zope           81 Dec 30  1998
standard_html_header
```

You can visit any of these "files" (which are really Zope objects) by selecting them in the usual Emacs way. Editing with Emacs is very useful, but for the most part, Emacs is a very complex program that is not very accessible to most people. Most Macintosh users, for example, would be very unfamiliar with a tool like Emacs. There are a number of "easier" editors that can be used that also use FTP and WebDAV. WebDAV is, in fact, designed to be used by tools like Adobe GoLive and Macromedia Dreamweaver.

Editing DTML Documents with WebDAV

WebDAV is a newer Internet protocol compared to HTTP or FTP, so there are fewer clients that support it. There is, however, a lot of momentum behind the WebDAV movement and more clients are being developed all the time. For more information on what programs support the WebDAV protocol, see the WebDAV homepage.

WebDAV is an extension to the HTTP protocol that provides rich features for many users concurrently authoring and editing content on web sites. WebDAV offers features like locking, revision control, and tagging documents or objects with properties. Because WebDAV's goals of through the web editing match some of the goals of Zope, Zope has supported the WebDAV protocol for quite a while.

The WebDAV protocol is evolving quickly, and new features are being added all the time. You can use any WebDAV client to edit your DTML Documents by simply pointing the client at your document's URL and editing it. For most clients, however, this will cause them to try to edit the *result* of rendering the document, not the *source*. For documents that use Zope's DTML template language to render dynamic content, this can be a problem.

Until clients catch up to the latest WebDAV standard and understand the difference between the source of a document and its result, Zope offers a special HTTP server you can enable with the `-W` command line option. This server listens on a different port than your normal HTTP server and returns different, special source content for WebDAV requests that come in on that port. This is an advanced feature and is explained more in the Documentation Section of Zope.org.

Using Zope Files

Zope Files contain raw data, just as the files on your computer do. Lots of information, like software, audio, video and documents are transported around the Internet and the world as files. You can use files to hold any kind of information that Zope doesn't specifically support, such as Flash files, applets, tarballs, etc.

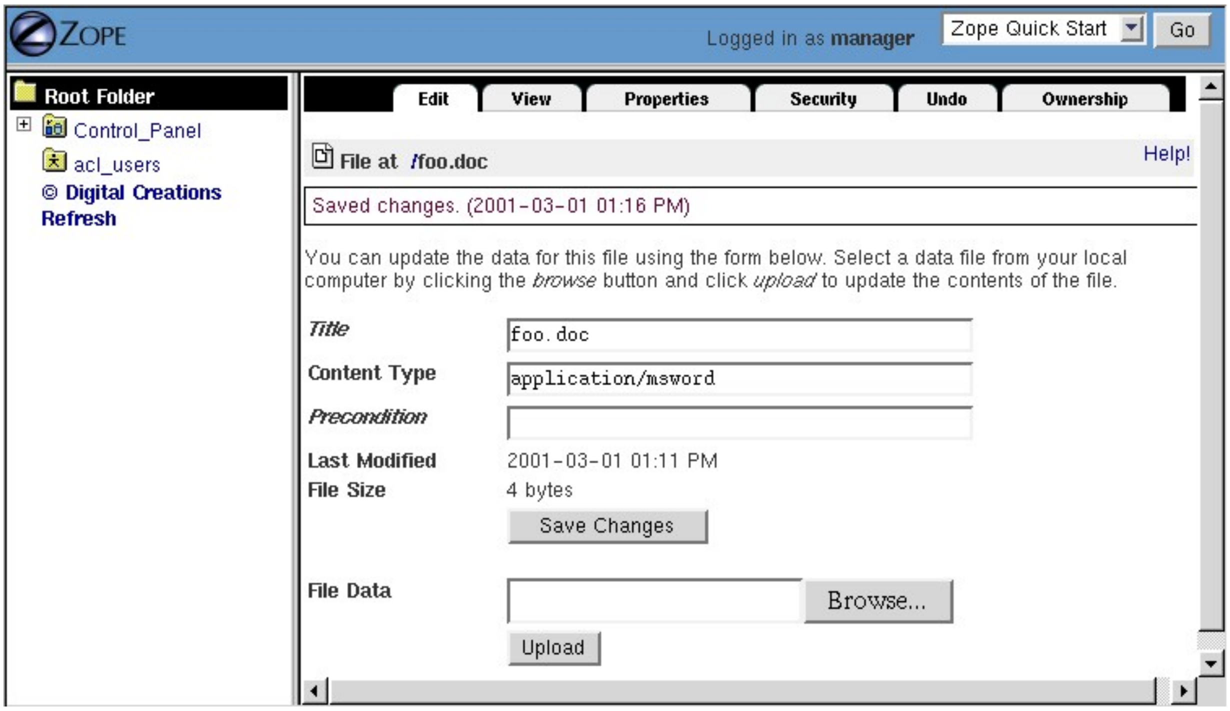
Files do not consider their contents to be of any special format, textual or otherwise. Files are good for holding any kind of *binary content* which is just raw computer information of some kind. Files are also good for holding textual content that doesn't need DTML scripting.

Every File object has a particular *content type* which is a standard Internet MIME designation for file type. When you upload a file into Zope, Zope tries to guess the content type from the name of the file, but Zope doesn't always guess correctly.

Uploading Files

Like DTML Documents and Methods, Files allow you to upload a file from your computer when you create a new object. Click the *Browse* button to choose a file from your local computer when creating a new Zope File. Try choosing a file such as a Word file (.doc) or a Portable Document Format (.pdf) file. Note, when uploading a file with your browser, you may have to indicate the file type you're looking for in your browser's upload dialog box. After selecting a file to upload, click *Add*. Depending on the size of the file you want to upload, it may take a few minutes to add the file to Zope.

After adding the File, click on the new File and look at its *Edit* view. Here you will see that Zope has guessed the content type as shown in Figure 3−5.



File at /foo.doc

Saved changes. (2001-03-01 01:16 PM)

You can update the data for this file using the form below. Select a data file from your local computer by clicking the *browse* button and click *upload* to update the contents of the file.

Title	foo.doc
Content Type	application/msword
Precondition	
Last Modified	2001-03-01 01:11 PM
File Size	4 bytes
File Data	

Figure 3−5 File content−type property

If you add a Word document, the content type is *application/msword*. If you add a PDF file, the content type is *application/pdf*. If Zope does not recognize the file type, it chooses the default, generic content type of *application/octet−stream*.

You can change the contents of an existing File by going to the *Upload* view. Here you can replace the contents of the File with a new file. If you don't fill in an id and title in this form and you upload a file, Zope will use the filename as the id and the title of the object.

Editing Files

If your File holds text and is less than 64K, then Zope lets you edit it in the management interface. A text file is one that has a content−type that starts with *text/*, such as *text/html*, or *text/plain*. You may sometimes find it convenient to edit text files in the management interface. In any case, you can always edit files locally and then upload them to Zope.

Viewing Files

You can view a file by going to the *View* tab from the management interface. You can also view a File by visiting its URL. In fact the *View* tab is just a way to get to a File's URL from the Zope management interface. For example, if you have a file in your Zope root folder called *employeeAgreement.pdf* then you can view that file in your web browser by going to the URL *http://localhost:8080/employeeAgreement.pdf*. Depending on the type of the file, your web browser may display the file or download it.

Using Zope Images

Images display graphics such as GIF, JPEG, and PNG files. In Zope, Images are similar to File objects, but include extra behavior for managing graphic content.

Image objects have the same management interface as file objects. Everything in the previous section about using file objects also applies to images. However, Image objects show you a preview of the image when you upload them.

Viewing Images with HTML

The most common use for Images in Zope is putting pictures in web pages. To put a picture into a web page, you need to use the HTML *IMG* tag. Suppose you have an Image object in your root folder called *logo* that contains an image of your organizations logo.

Using this Image in your HTML is a straight forward process: you can reference it with an *IMG* tag as you'd do to include any type of image in a web page:

```
<dtml-var standard_html_header>

  <img src="logo">

  <h1>Welcome!</h1>

<dtml-var standard_html_footer>
```

In this example, you reference the logo image by creating an HTML *IMG* tag, but usually it is not necessary to create your own *IMG* tags to display images. Image objects know how to generate their own HTML tags. When you insert an Image object in DTML, it generates an *IMG* tag for itself.

Now, we want this logo to be seen on every page up in the upper left−hand corner, so put a reference to it in the *standard_html_header* method:

```
<html>
  <body>
    <dtml-var logo>
```

Now, view the root folder by clicking on the *View* tab. If you look at the source to the web page that Zope creates, you can see that the `var` DTML code was turned into an HTML *IMG* tag for you:

```
<html>
  <body>
    <img src="logo" width="50"  height="30">
```

Using the DTML *var* tag to draw Images makes things simple, because Zope automatically figures out the height and width attributes of the *IMG* tag for you. If you don't like the way Zope constructs an *IMG* tag, it can be customized. See Appendix B for more information on the Image object and how it can control the *IMG* tag.

There are a number third party Zope object types (generally called "Products") for storing and viewing image content available from the visual section of Zope.org.

Viewing Images Through the Web

Images can be viewed directly by going to their URL in your web browser. For example, let's say you want to view your company logo directly. The logo exists as an image object in your root folder. It is called *logo*, you can easily view it by going directly to its URL *http://localhost:8080/logo*.

Since Zope Images work just like images stored in a normal web server, you can access your Zope images from other web servers. Suppose you have a Zope Image whose URL is *http://imageserver:8080/Birds/Parakeet.jpg*. You can include this Image in any web page served from any web server using the Image's absolute URL in your web page:

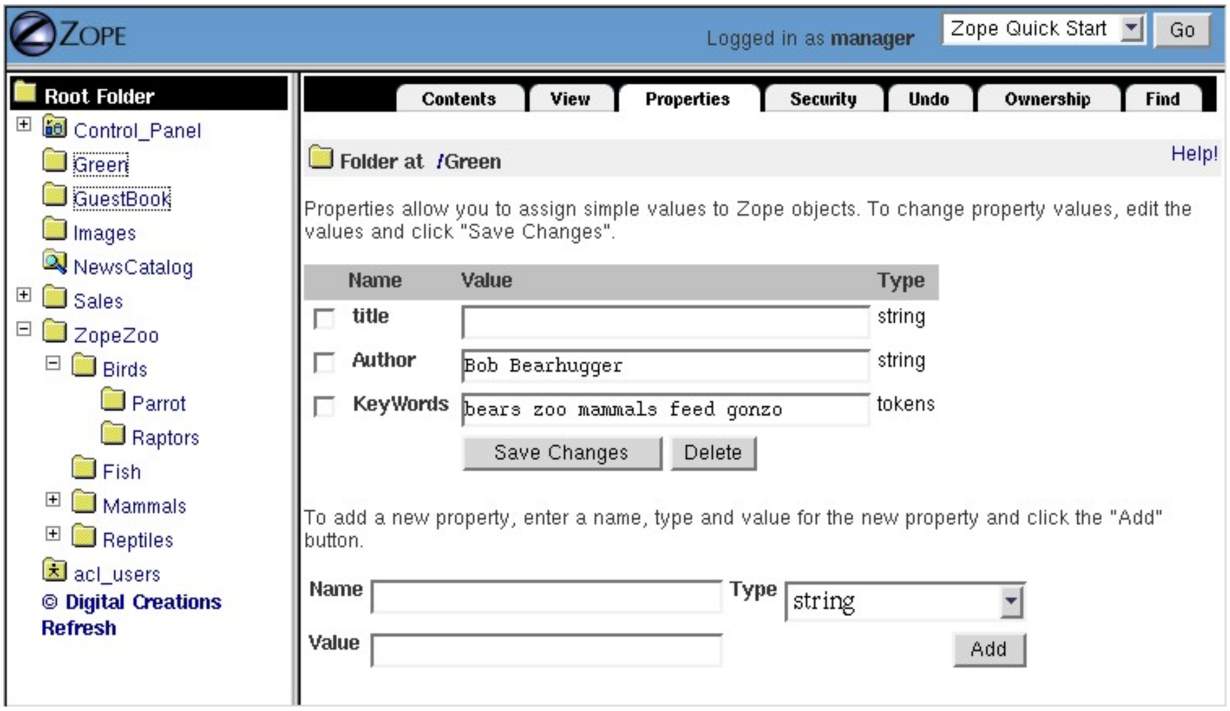
```
<html>

<h1>Remote Image</h1>

<img src="http://imageserver:8080/Birds/Parakeet.jpg">

</html>
```

This example shows how you can use Zope data from outside Zope using standard Internet protocols. Later in Chapter 10, "Advanced Zope Scripting" you'll see how most Zope objects can provide services to the outside world.

Using Object Properties

Properties are ways of associating information with objects in Zope. Many Zope objects, including folders and documents, support properties. Properties can label an object in order to identify its contents (many Zope content objects have a content type property). Another use for properties is to provide meta−data for an object such as its author, title, status, etc.

Properties can be more complex than strings; they can also be numbers, lists, or other data structures. All properties are managed via the *Properties* view. Click on an object's *Properties* tab and you will be taken to the properties management view, as seen in Figure 3−6.



Folder at /Green

Properties allow you to assign simple values to Zope objects. To change property values, edit the values and click "Save Changes".

Name	Value	Type
title		string
Author	Bob Bearhugger	string
KeyWords	bears zoo mammals feed gonzo	tokens

To add a new property, enter a name, type and value for the new property and click the "Add" button.

Name: Type: string
Value:

Figure 3−6: The Properties Management View

A property consists of a name, a value and a type. A property's type defines what kind of value or values it can have.

In Figure 3−6 you can see that the folder has three properties, *title*, *Author*, *KeyWords*. The *title* and *Author* property are *string* properties, while the *KeyWords* property has a type of *tokens*. A *tokens* property is like a sequence of words.

Zope supports a number of property types. Each type is suited to a specific task. This list gives a brief overview of the kinds of properties you can create from the management interface:

string	A string is an arbitrary length sequence of characters. Strings are the most basic and useful type of property in Zope.
int	An int property is an integer, which can be any positive or negative number that is not a fraction. An int is guaranteed at least 32 bits long.
long	A long is like an integer that has no range limitation.
float	A float holds a floating point, or decimal number. Monetary values, for example, often use floats.
lines	A lines property is a sequence of strings.
tokens	A tokens property is list of words separated by spaces.
text	A text property is just like a string property, except that Zope normalizes the line ending characters (different browsers use different line ending conventions).
selection	A selection property is special, it is used to render an HTML select input widget.
multiple selection	A multiple selection property is special, it is used to render an HTML multiple select form input widget.

Properties are very useful tools for tagging your Zope objects with little bits of data or information. In conjunction with methods and scripts, properties make extending simple objects like Folders a very powerful technique.

Coding Logic with Scripts

In traditional programming lingo, a *script* is a short piece of code written in a programming language. As of version 2.3, Zope now comes with two kinds of script objects: one that lets you write scripts in Python and one that lets you write scripts in Perl.

Both Python and Perl are very popular and powerful programming languages. Both Python and Perl share many similar feature: both offer powerful, rapid development, simple syntax, many add−on libraries, strong community following, and copious amounts of free, online documentation. Both languages are also open source.

Because scripts are so powerful and flexible, their possible uses are endless. Scripts are primarily used to write what is called *business logic*. Business logic is different than presentation logic. Presentation logic is usually written in a presentation language, like DTML, and its purpose is to display information to a user. Business logic is usually written in a scripting language, and its purpose is to manipulate information that comes from content sources (like documents or databases) or manipulate other objects. Often, presentation logic is based *on top of* business logic.

A simple example of using scripts is building an online web form to help your users calculate the amount of compound interest on their debts. This kind of calculation involves the following procedure:

1. You need the following information: your current account balance (or debt) called the "principal", the annual interest rate expressed as a decimal (like 0.095) called the "interest_rate", the number of times during the year interest in compounded (usually monthly), called the "periods" and the number of years from now you want to calculate, called the "years" .
2. Divide your "interest_rate" by "periods" (usually 12). We'll call this result "i".
3. Take "periods" and multiply it by "years". We'll call this result "n".
4. Raise (1 + "i") to the power "n".
5. Multiply the result by your "principal". This is the new balance (or debt).

For this example, you will need two page templates named *interestRateForm* and *interestRateDisplay* to collect the information from the user and display it, respectively. You will also need a Python−based script called *calculateCompoundingInterest* that will do the actual calculation. The first step is to create a web form in *interestRateForm* that collects "principal", "interest_rate", "periods" and "years" from your users. Here's an example *interestRateForm* page templates:

```
<html>
  <body>

  <form action="interestRateDisplay" method="POST">
  <p>Please enter the following information:</p>

  Your current balance (or debt): <input name="principal:float"><br>
  Your annual interest rate: <input name="interest_rate:float"><br>
  Number of periods in a year: <input name="periods:int"><br>
  Number of years: <input name="years:int"><br>
  <input type="submit" value=" Calculate "><br>
  </form>

  </body>
</html>
```

This form collects information and calls the *interestRateDisplay* template. Now, create a Python−based script called *calculateCompoundingInterest* that accepts four parameters, "principal", "interest_rate", "periods" and "years" with the following python code:

```
## Script (Python) "calculateCompoundInterest"
##parameters=principal, interest_rate, periods, years
##
"""
Calculate compounding interest.
"""
i = interest_rate / periods
n = periods * years
return ((1 + i) ** n) * principal
```

Enter the parameters into the *Parameters List* field, and the code in the body text area. The comments shown at the beginning of the code are not necessary when editing through the web. (However these comments are useful for editing scripts via FTP.)

This will return the balance or debt compounded over the course of "years". Next, create a *interestRateDisplay* page template that calls *calculateCompoundingInterest* and returns the result:

```
<html>
  <body>
```

```
    <p>Your total balance (or debt) including compounded interest over
    <span tal:content="years">2</span> years is:</p>
    <p><b>$<span tal:content="python: here.calculateCompoundingInterest(principal,
                                                       interest_rate,
                                                       periods,
                                                       years)" >1.00</span></b></p>

    </body>
  </html>
```

First view the *interestRateForm* page template. Now, type in some information about your balance or debt and click *Calculate*. This will cause *interestRateForm* to submit the collect information to *interestRateDisplay*, which calls the Python−based script *calculateCompoundingInterest*. The display method uses the value returned by the script in the resulting display.

As we said earlier, the possibilities for using scripts is almost endless. This example, however, gives you a good idea of the most common pattern for *presentation* objects to collect and display information, and using *business logic* objects to make calculations.

Using Methods

Methods are objects in Zope that hold special executable content. The name "Method" is actually a bit of a misnomer, and its use in Zope is slowly being phased out for more common terms like *Script* and *Template*.

Zope comes with two kinds of methods, DTML Methods and SQL Methods. DTML Methods are used to define *presentation templates* that you can apply to content objects like DTML Documents and Files. A very common and popular way to use DTML Methods is to define presentation layout separate from your content.

SQL Methods are used to contain database queries that you can reuse throughout your web application. SQL Methods are explained in Chapter 12, "Relational Database Connectivity", where an example of creating a web application using a relational database is given.

All the various objects in Zope can be manipulated by calling methods on those objects. For example, Folder objects have an `objectValues` method that returns the objects contained by the folder. DTML Methods can be used to write simple scripts that call these Zope API methods. These methods are documented in the Help System, under API Documentation.

Before Zope 2.3, DTML Methods were the only way to write scripts in Zope with your web browser. While DTML is useful for very simple scripts and for presenting information with templates, this approach had a number of limitations because DTML isn't as flexible as other programming languages.

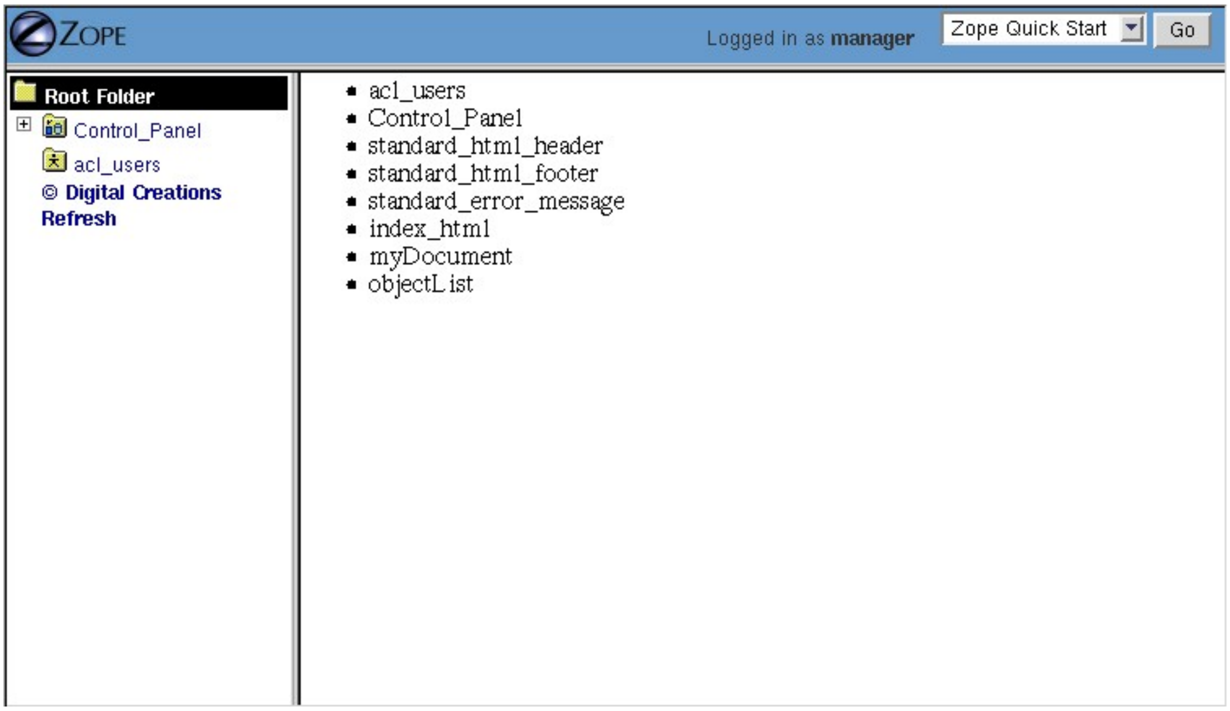
Zope 2.3 introduces two new kinds of *Script* objects based on two very popular programming languages, Python (which Zope is written in) and Perl. You should use Python and Perl−based scripts to write more complex scripts instead of a DTML Method. While browsing through past Zope documentation, mail list archives, and other resources on "Zope.org"http://www.zope.org, you may find a lot of references to very complex DTML scripts. These pre−date Python and Perl−based scripts. In general, complex scripts should be written in either Python or Perl. Python and Perl−based scripts are described later in this chapter, and many examples of their use is given in Chapter 10, "Advanced Zope Scripting".

A simple example of using DTML Methods is to create a DTML Method in the root folder called *objectList*:

```
<dtml−var standard_html_header>

<ul>
```

```
    <dtml-in objectValues>
      <li><dtml-var getId></li>
    </dtml-in>
  </ul>

  <dtml-var standard_html_footer>
```

When you view this method, it calls the *objectValues* method on the root folder and this shows you a simple HTML list of all the objects in the root folder, as shown in Figure 3−7.



- acl_users
- Control_Panel
- standard_html_header
- standard_html_footer
- standard_error_message
- index_html
- myDocument
- objectList

Figure 3−7 Results of the objectList DTML Method

All folders implement the *objectValues* method. The *objectValues* method is part of an interface that all folders implement called *ObjectManager*.

In addition to calling API methods on objects, DTML Methods can also be used in a certain way to extend any Zope object. This will be explained in more detail in the next chapter. In effect, this allows you to extend the Zope API by simply creating DTML Methods.

You just saw the *objectList* method, which resides in the root folder, and makes a simple list of the contents of the root folder. Because the method is in the root folder, it is now usable by any other objects in or below the root folder. This method extends the Zope API for these objects since it provides them with another callable method.

To demonstrate, let's create a subfolder called *Primates* and add three documents, *Monkeys*, *Apes*, *Humans*. You can call the *objectList* method on the *Primates* folder by visiting the URL *Primates/objectList*. You can see the effect of calling the *objectList* method on the *Primates* folder differs from the effect of calling it on the root folder. The *objectList* method is defined in the root folder, but here we are using it to display the contents of the *Primates* folder. This mechanism of reusing objects is called *acquisition* and will be explained more in Chapter 4, "Dynamic Content with DTML".

DTML Methods mainly serve as presentation templates. DTML Methods can act as templates tying reusable bits of content together into dynamic web pages. The template features of DTML Methods will be discussed in further detail in the next chapter.

Comparing DTML Documents and Methods

DTML Methods have the same user interface as DTML Documents, which can be a bit confusing to the beginner. All of the procedures that you learned in the last chapter for adding, editing, viewing and uploading DTML Documents are identical for DTML Methods.

A source of frequent confusion for Zope beginners is the question of when to use a DTML Document versus when to use a DTML Method. On the surface, these two options seem identical. They both hold DTML and other content, they both execute DTML code, and they both have a similar user interface and a similar API, so what's the difference?

DTML Documents are meant to hold document−like content. For example, the various chapters of a book could be held in a DTML Document. A general rule is: if your content is mostly document−like and you want to present it on your site, then it should go into a DTML Document.

DTML Methods are meant to manipulate and display other objects. DTML Methods don't usually hold a lot of content, unless the content is meant to change or manipulate other content.

Don't worry if you're still unclear on the differences between DTML Document and Methods. Even the most experienced Zope programmers need to think a little before deciding which type of object to use. In Chapter 8, "Variables and Advanced DTML", you'll learn about the technical differences between DTML Documents and DTML Methods (they look up variables differently since they have different "client" objects). Here are some general rules to help you decide between DTML Documents and Methods:

- If it's content, use a DTML Document, or a File if it doesn't require any DTML scripting.
- If it's simple logic, use a DTML Method.
- If it's meant to be presented by other objects, use a DTML Document.
- If it's meant to present other objects, use a DTML Method.
- If it's complex behavior, use a Python or Perl−based script.

As you've seen DTML Methods are a useful tool for presentation and quick scripting, but eventually you're going to want to power of a fully expressive programming language, and that's where *Scripts* come in.

Using Sessions

Sessions allow you keep track of site visitors. Web browsers use a protocol named HTTP to exchange data with a server such as Zope. HTTP is does not provide a way for the server to keep track of a user's requests; each request is considered completely independent.

Sessions overcome this limitation of HTTP. The term "session" means a series of related HTTP requests that come from the same client during a given time period. Zope's sessioning system makes use of cookies and/or HTTP form elements "in the background" to keep track of user sessions. Zope's sessioning system allows you to avoid manually managing user sessions.

You can use sessions to keep track of anonymous users as well as those who have Zope login accounts.

Data associated with a session is called "session data". Session data is valid only for the duration of one site visit as determined by a configurable inactivity timeout value. Session data is used to keep track of information about a user's visit such as the items that a user has put into a "shopping cart", or which pages a user has seen on his trip to your site.

It is important to realize that keeping sensitive data in a session data object is potentially insecure unless the connection between browsers and Zope is encrypted in some way. Don't store sensitive information such as phone numbers, addresses, account numbers, credit card numbers or any other personal information about your site visitors unless you've secured the connection between Zope and site visitors via SSL.

Session Configuration

Zope versions after 2.5 come with a default sessioning environment configured "out of the box", so there's no need to change these objects unless you're curious or want to change how sessions are configured. For information on changing sessioning configuration, use the Zope help system.

Zope uses several different types of objects to manage session data, and brief explanations of their purpose follow.

Browser ID Manager
: This object manages how visitors' browsers are identified from request−to−request, and allows you to configure whether this happens via cookies or form variables, or via a combination of both. The default sessioning configuration provides a Browser Id Manager as the `/browser_id_manager` object.

Transient Object Container
: This object holds session data. It allows you to set how long session data lasts before it expires. The default sessioning configuration provides a Transient Object Container named `/temp_folder/session_data`. The session data objects in the default `session_data` Transient Object container are lost each time Zope is restarted.

Session Data Manager
: This object connects the browser id and session data information. When a folder which contains a session data manager is traversed, the REQUEST object is populated with the SESSION, which is a session data object. The default sessioning configuration provides a Session Data Manager named `/session_data_manager`.

Using Session Data

You will typically access session data through the `SESSION` attribute of the REQUEST object.

Here's an example of how to work with a session using a Python−based Script:

```
## Script (Python) "lastView"
secs_per_day=24*60*60
session=context.REQUEST.SESSION
if session.has_key('last view'):
    # The script has been viewed before, since the 'last view'
    # has been previously set in the session.
    then=session['last view']
    now=context.ZopeTime()
    session['last view']=now # reset last view to now
    return 'Seconds since last view %.2f' % ((now − then) * secs_per_day)
# The script hasn't been viewed before, since there's no 'last
# view' in the session data.
session['last view']=context.ZopeTime()
return 'This is your first view'
```

View this script, and then reload it a couple of times. It keeps track of when you last viewed the script and calculates how long it has been since you last viewed it. Notice that if you quit your browser and come back to the script it forgets you. However, if you simply visit some other pages and then return, it still remembers the last time you viewed it.

This example shows the basic features of working with session data: session data objects act like Python dictionaries. You will almost always use session data that consists of normal Python lists, dictionaries, strings, and numbers. The only tricky thing about sessions is that when working with mutable session data (for example dictionaries or lists) you need to save the session data by reassigning it. Here's an example:

```
## Script (Python) "sessionExample"
session=context.REQUEST.SESSION
# l is a list
l=session['myList']
l.append('spam')
# If you quit here, your changes to the list won't
# be saved. You need to save the session data by
# reassigning it to the session.
session['myList']=l
```

For more information about persistence and mutable data, see the *Zope Developer's Guide*.

You can use sessions in Page Templates and DTML Documents, too. For example, here's a template snippet that displays the users favorite color (as stored in a session):

```
<p tal:content="request/SESSION/favorite_color">Blue</p>
```

Here's how to do the same thing in DTML:

```
<dtml-with SESSION mapping>
  <p><dtml-var favorite_color></p>
</dtml-with>
```

Sessions have a plethora of additional configuration parameters and usage patterns. For further information about the session application programming interface, see the Zope help system. For an additional example of using sessions, see the "shopping cart" example that comes with Zope 2.5 and above (in the Examples folder).

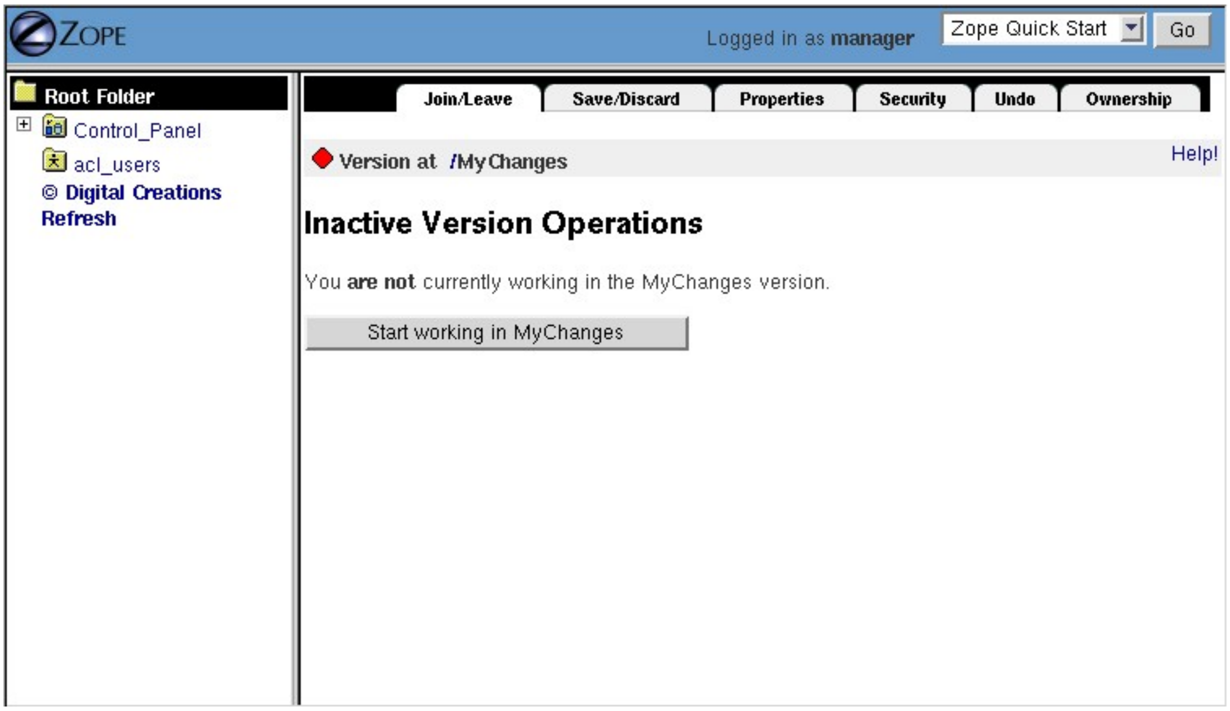
Using Versions

Version objects help coordinate the work of many people on the same set of objects. While you are editing a document, someone else can be editing another document at the same time. In a large Zope site hundreds or even thousands of people can be using Zope simultaneously. For the most part this works well, but problems can occur. For example, two people might edit the same document at the same time. When the first person finishes their changes they are saved in Zope. When the second person finishes their changes they over write the first person's changes. You can always work around this problem using *Undo* and *History*, but it can still be a problem. To solve this problem, Zope has *Version* objects.

Another problem that you may encounter is that you may wish to make some changes, but you may not want to make them public until you are done. For example, suppose you want to change the menu structure of your site. You don't want to work on these changes while folks are using your site because it may break the navigation system temporarily while you're working.

Versions are a way of making private changes in Zope. You can make changes to many different documents without other people seeing them. When you decide that you are done you can choose to make your changes public, or discard them. You can work in a Version for as long as you wish. For example it may take you a week to put the finishing touches on your new menu system. Once you're done you can make all your changes live at once by committing the version.

Create a Version by choosing Version from the product add list. You should be taken to an add form. Give your Version an id of *MyChanges* and click the *Add* button. Now you have created a version, but you are not yet using it. To use your version click on it. You should be taken to the *Join/Leave* view of your version as shown in Figure 3−8.



Version at /MyChanges

Inactive Version Operations

You are not currently working in the MyChanges version.

Start working in MyChanges

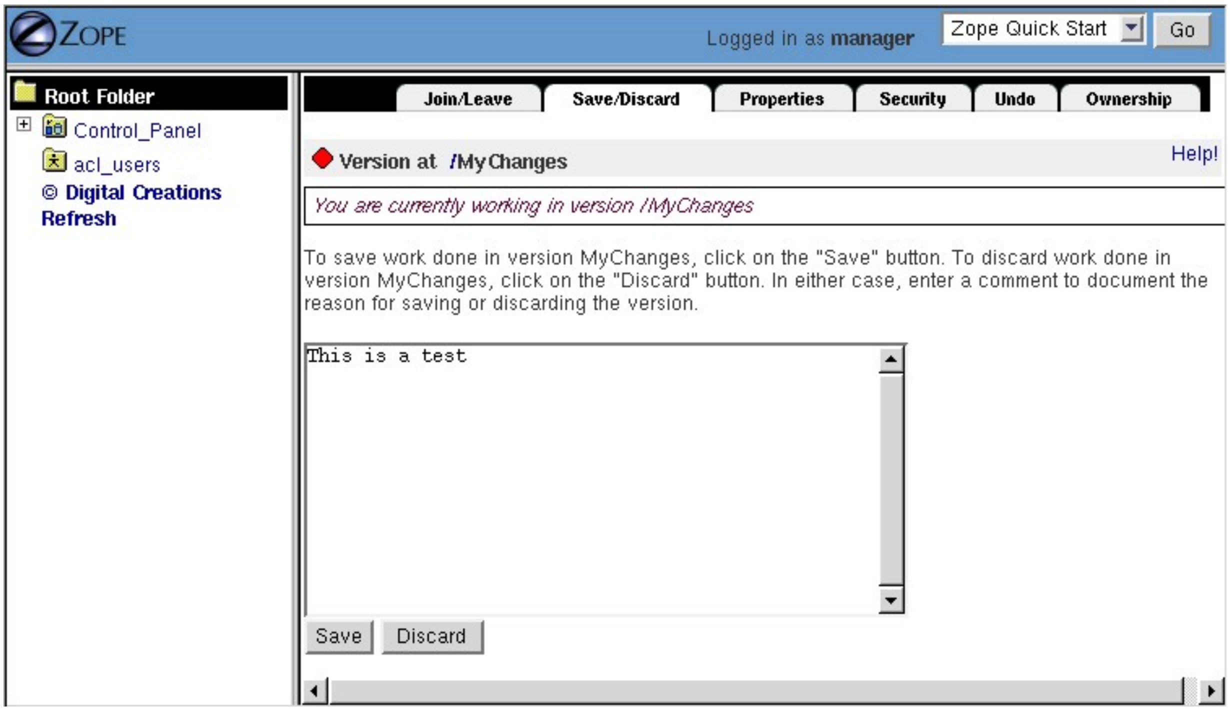
Figure 3−8 Joining a Version

The Version is telling you that you are not currently using it. Click on the *Start Working in MyChanges* button. Now Zope should tell you that you are working in a version. Now return to the root folder. Notice that everywhere you go you see a small message at the top of the screen that says *You are currently working in version /MyChanges*. This message lets you know that any changes you make at this point will not be public, but will be stored in your version. For example, create a new DTML Document named *new*. Notice how it has a small red diamond after its id. Now edit your *standard_html_header* method. Add a line to it like so:

```
<HTML>
  <HEAD>
    <TITLE><dtml-var title_or_id></TITLE>
  </HEAD>
  <BODY BGCOLOR="#FFFFFF">
  <H1>Changed in a Version</H1>
```

Any object that you create or edit while working in a version will be marked with a red diamond. Now return to your version and click the *Quit working in MyChanges* button. Now try to return to the *new* document. Notice that the document you created while in your version has now disappeared. Any other changes that you made in the version are also gone. Notice how your *standard_html_header* method now has a small red diamond and a lock symbol after it. This indicates that this object has been changed in a version. Changing an object in a version locks it, so no one else can change it until you commit or discard the changes you made in your version. Locking ensures that your version changes don't overwrite changes that other people make while you're working in a version. So for example if you want to make sure that only you are working on an object at a given time you can change it in a version. In addition to protecting you from unexpected changes, locking also makes things inconvenient if you want to edit something that is locked by someone else. It's a good idea to limit your use of versions to avoid locking other people out of making changes to objects.

Now return to your version by clicking on it and then clicking the *Start working in MyChanges* button. Notice how everything returns to the way it was when you left the Version. At this point let's make your changes permanent. Go to the *Save/Discard* view as shown in Figure 3−9.



Version at /MyChanges

You are currently working in version /MyChanges

To save work done in version MyChanges, click on the "Save" button. To discard work done in version MyChanges, click on the "Discard" button. In either case, enter a comment to document the reason for saving or discarding the version.

```
This is a test
```

Figure 3−9 Committing Version changes.

Enter a comment like *This is a test* into the comment field and click the *Save* button. Your changes are now public, and all objects that you changed in your Version are now unlocked. Notice that you are still working in your Version. Go to the *Join/Leave* view and click the *Quit Working in MyChanges* button. Now verify that the document you created in your version is visible. Your change to the *standard_html_header* should also be visible. Like anything else in Zope you can choose to undo these changes if you want. Go to the *Undo* view. Notice that instead of many transactions one for each change, you only have one transaction for all the changes you made in your version. If you undo the transaction, all the changes you made in the version will be undone.

Versions are a powerful tool for group collaboration. You don't have to run a live server and a test server since versions let you make experiments, evaluate them and then make them public when you decide that all is well. You are not limited to working in a version alone. Many people can work in the same version. This way you can collaborate on version's changes together, while keeping the changes hidden from the general public.

Versions and ZCatalog

Versions don't work well with ZCatalog. This is because versions lock objects when they are modified in a version, preventing changes outside the version. This works well when changes are isolated.

ZCatalog has a way of connecting changes made to disparate objects. This is because cataloging an object must, by necessity change the catalog. Objects that automatically catalog themselves when they are changed propigate their changes to the catalog. If such an object is changed in a version, then the catalog is changed in the version too, thus locking the catalog. This property makes the catalog and versions get along poorly. As a rule, versions should not be used in applications that use the catalog.

Improving Performance with Caching

A *cache* is a temporary place to store information that you access frequently. The reason for using a cache is speed. Any kind of dynamic content, like a DTML page or a Python Script, must be evaluated each time it is called. For simple pages or quick scripts, this is usually not a problem. For very complex DTML pages or scripts that do a lot of computation or call remote servers, accessing that page or script could take more than a trivial amount of time. Both DTML and Python can get this complex, especially if you use lots of looping (such as the `in` tag or the Python `for` loop) or if you call lots of scripts, that in turn call lots of scripts, and so on. Computations that take a lot of time are said to be *expensive*.

A cache can add a lot of speed to your site by calling an expensive page or script once and storing the result of that call so that it can be reused. The very first person to call that page will get the usual slow response time, but then once the value of the computation is stored in the cache, all subsequent users to call that page will see a very quick response time because they are getting the *cached copy* of the result and not actually going through the same expensive computation the first user went through.

To give you an idea of how caches can improve your site speed, imagine that you are creating *www.zopezoo.org*, and that the very first page of your site is very complex. Let's suppose this page has complex headers, footers, queries several different database tables, and calls several special scripts that parse the results of the database queries in complex ways. Every time a user comes to *www.zopezoo.org*, Zope must render this very complex page. For the purposes of demonstration, let's suppose this complex page takes one−half of a second, or 500 milliseconds, to compute.

Given that it takes a half of a second to render this fictional complex main page, your machine can only really serve 120 hits per minute. In reality, this number would probably be even lower than that, because Zope has to do other things in addition to just serving up this main page. Now, imagine that you set this page up to be cached. Since none of the expensive computation needs to be done to show the cached copy of the page, many more users could see the main page. If it takes, for example, 10 milliseconds to show a cached page, then this page is being served *50 times faster* to your web site visitors. The actual performance of the cache and Zope depends a lot on your computer and your application, but this example gives you an idea of how caching can speed up your web site quite a bit. There are some disadvantages to caching however:

Cache lifetime
: If pages are cached for a long time, they may not reflect the most current information on your site. If you have information that changes very quickly, caching may hide the new information from your users because the cached copy contains the old information. How long a result remains cached is called the *cache lifetime* of the information.

Personal information
: Many web pages may be personalized for one particular user. Obviously, caching this information and showing it to another user would be bad due to privacy concerns, and because the other user would not be getting information about *them*, they'd be getting it about someone else. For this reason, caching is often never used for personalized information.

Zope allows you to get around these problems by setting up a *cache policy*. The cache policy allows you to control how content gets cached. Cache policies are controlled by *Cache Manager* objects.

Adding a Cache Manager

Cache managers can be added just like any other Zope object. Currently Zope comes with two kinds of cache managers:

HTTP Accelerated Cache Manager

An HTTP Accelerated Cache Manager allows you to control an HTTP cache server that is external to Zope, for example, Squid. HTTP Accelerated Cache Managers do not do the caching themselves, but rather set special HTTP headers that tell an external cache server what to cache. Setting up an external caching server like Squid is beyond the scope of this book, see the Squid site for more details.

(RAM) Cache Manager

A RAM Cache Manager is a Zope cache manager that caches the content of objects in your computer memory. This makes it very fast, but also causes Zope to consume more of your computer's memory. A RAM Cache Manager does not require any external resources like a Squid server, to work.

For the purposes of this example, create a RAM Cache Manager in the root folder called *CacheManager*. This is going to be the cache manager object for your whole site.

Now, you can click on *CacheManager* and see its configuration screen. There are a number of elements on this screen:

Title

The title of the cache manager. This is optional.

REQUEST variables

This information is used to store the cached copy of a page. This is an advanced feature, for now, you can leave this set to just "AUTHENTICATED_USER".

Threshold Entries

The number of objects the cache manager will cache at one time.

Cleanup Interval

The lifetime of cached results.

For now, leave all of these entries as is, they are good, reasonable defaults. That's all there is to setting up a cache manager!

There are a couple more views on a cache manager that you may find useful. The first is the *Statistics* view. This view shows you the number of cache "hits" and "misses" to tell you how effective your caching is.

There is also an *Associate* view that allows you to associate a specific type or types of Zope objects with a particular cache manager. For example, you may only want your cache manager to cache DTML Documents. You can change these settings on the *Associate* view.

At this point, nothing is cached yet, you have just created a cache manager. The next section explains how you can cache the contents of actual documents.

Caching a Document

Caching a document is very easy. First, before you can cache a document you must have a cache manager like the one you created in the previous section.

To cache a document, create a new DTML Document object in the root folder called *Weather*. This object will contain some weather information. For example, let's say it contains:

```
<dtml-var standard_html_header>

  <p>Yesterday it rained.</p>

<dtml-var standard_html_footer>
```

Now, click on the *Weather* DTML Document and click on its *Cache* view. This view lets you associate this document with a cache manager. If you pull down the select box at the top of the view, you'll see the cache manager you created in the previous section, *CacheManager*. Select this as the cache manager for *Weather*.

Now, whenever anyone visits the *Weather* document, they will get the cached copy instead. For a document as trivial as our *Weather* example, this is not much of a benefit. But imagine for a moment that *Weather* contained some database queries. For example:

```
<dtml-var standard_html_header>

  <p>Yesterday's weather was <dtml-var yesterdayQuery> </p>

  <p>The current temperature is <dtml-var currentTempQuery></p>

<dtml-var standard_html_footer>
```

Let's suppose that *yesterdayQuery* and *currentTempQuery* are SQL Methods that query a database for yesterdays forecast and the current temperature, respectively (for more information on SQL Methods, see Chapter 12, "Relational Database Connectivity"). Let's also suppose that the information in the database only changes once every hour.

Now, without caching, the *Weather* document would query the database every time it was viewed. If the *Weather* document was viewed hundreds of times in an hour, then all of those hundreds of queries would always contain the same information.

If you specify that the document should be cached, however, then the document will only make the query when the cache expires. The default cache time is 300 seconds (5 minutes), so setting this document up to be cached will save you 91% of your database queries by doing them only one twelfth as often. There is a trade−off with this method, there is a chance that the data may be five minutes out of date, but this is usually an acceptable compromise.

For more information about caching and using the more advanced options of caching, see the Zope Administrator's Guide.

Virtual Hosting Objects

Zope comes with three objects that help you do virtual hosting, *SiteRoot*, *Set Access Rule*, and *Virtual Host Monster*. Virtual hosting is a way to serve many web sites with one Zope server. Virtual hosting is an advanced administration function, that is beyond the scope of this book. See the Zope Administrator's Guide for more information on virtual hosting.

Sending mail with MailHost

Zope comes with an object that is used to send outbound e−mail, usually in conjunction with the DTML `sendmail` tag, described more in Chapter 8, "Variables and Advanced DTML".

Mailhosts can be used from either Python or DTML to send an email message over the Internet. They are useful as `gateways` out to the world. Each mailhost object is associated with one mail server, for example, you can associate a mailhost object with `yourmail.yourdomain.com`, which would be your outbound SMTP mail server. Once you associate a server with a mailhost object, the mailhost object will always use that server to send mail.

To create a mailhost object select *MailHost* from the add list. You can see that the default id is "MailHost" and the default SMTP server and port are "localhost" and "25". make sure that either your localhost machine is running a mail server, or change "localhost" to be the name of your outgoing SMTP server.

Now you can use the new MailHost object from a DTML `sendmail` tag. This is explained in more detail in Chapter 8, "Variables and Advanced DTML". The API for MailHost objects also allows you to send mail from Python scripts. For more information, see the online help system.

Chapter 4: Dynamic Content with DTML

DTML (Document Template Markup Language) is Zope's tag−based presentation and scripting language. DTML dynamically generates, controls, and formats content. DTML is commonly used to build modular and dynamic web interfaces for your web applications.

DTML is a server side scripting language, like SSI, PHP, ASP, and JSP. This means that DTML commands are executed by Zope at the server, and the result of that execution is sent to your web browser. By contrast, client−side scripting languages like Javascript are not processed by the server, but are rather sent to and executed by your web browser.

You can use DTML scripting in two types of Zope objects, *DTML Documents* and *DTML Methods*.

Who is DTML For?

DTML is designed for people familiar with HTML and basic web scripting, not for application programmers. In fact, if you want to do programming with Zope you shouldn't use DTML. In Chapter 9, "Advanced Zope Scripting", we'll cover advanced programming using Python and Perl.

DTML is for presentation and should be managed by web designers. Zope encourages you to keep your presentation and logic separate by providing different objects for presentation (DTML), and logic (Python, Perl, and others). You will find a host of benefits resulting from keeping your presentation in DTML and your logic in other types of Zope objects. Some of those benefits include:

- Keeping logic and presentation separate makes it easy to vary either component without disrupting the other.
- Often you will have different people in charge of maintaining logic and presentation. By using different objects for these tasks you make it easier for people to collaborate without disrupting each other.
- It's easier to reuse existing presentation and logic components if they are not intermingled.

What is DTML Good for?

DTML is good for creating dynamic web interfaces. It supports reusing content and layout, formatting heterogeneous data, and separating presentation from logic and data.

For example with DTML you can reuse shared web page headers and footers:

```
<dtml−var standard_html_header>

<p>Hello world.</p>

<dtml−var standard_html_footer>
```

This web page mixes HTML and DTML together. DTML commands are written as tags that begin with *dtml−*. This example builds a web page by inserting a standard header and footer into an HTML page. The resulting HTML page might look something like this:

```
<html>
<body bgcolor="#FFFFFF">

<p>Hello world.</p>
```

```
<hr>
<p>Last modified 2000/10/16 by AmosL</p>
</body>
</html>
```

As you can see the standard header defined a white background color and the standard footer added a note at the bottom of the page telling when the page was last modified and by whom.

In addition to reusing content, DTML lets you easily and powerfully format all kinds of data. You can use DTML to call methods, query databases, introspect Zope objects, process forms, and more.

For example when you query a database with a SQL Method it typically returns a list of results. Here's how you might use DTML to format each result from a database query:

```
<ul>
<dtml-in frogQuery>
  <li><dtml-var animal_name></li>
</dtml-in>
</ul>
```

The DTML *in* tag iterates over the results of the database query and formats each result. Suppose four results are returned by *frogQuery*. Here's what the resulting HTML might look like:

```
<ul>
  <li>Fire-bellied toad</li>
  <li>African clawed frog</li>
  <li>Lake Nabu reed frog</li>
  <li>Chilean four-eyed frog</li>
</ul>
```

The results of the database query are formatted as an HTML bulleted list.

Note that you don't have to tell DTML that you are querying a database and you don't have to tell it where to find the arguments to call the database query. You just tell it what object to call, it will do the work of figuring out how to call the object and pass it appropriate arguments. If you replace the *frogQuery* SQL Method with some other kind of object, like a Script, a ZCatalog, or even another DTML Method, you won't have to change the way you format the results.

This ability to format all kinds of data makes DTML a powerful presentation tool, and lets you modify your business logic without changing your presentation.

When Not to Use DTML

DTML is not a general purpose programming language. For example, DTML does not allow you to create variables very easily. While it may be possible to implement complex algorithms in DTML, it is painful and not recommended. If you want to implement programming logic, use Python or Perl (for more information on these subjects, see Chapter 9, "Advanced Zope Scripting").

For example, let's suppose you were writing a simple web page for a group of math students, and on that page you wanted to illustrate a simple calculation. You would not want to write the program that made this calculation in DTML. It could be *done* in DTML, but it would be difficult to understand. DTML would be perfect for describing the page that this calculation is inserted into, but it would be awful to do this calculation in DTML, whereas it may be very simple and trivial in Python or Perl.

String processing is another area where DTML is not the best choice. If you want to manipulate input from a user in a complex way, but using functions that manipulate strings, you are better off doing it in Python or Perl, both of which have much more powerful string processing abilities than DTML.

DTML is one tool among many available in Zope. If you find yourself scratching your head trying to figure out some complicated DTML construct, there's a good chance that things would work better if you broke your DTML script up into a collection of DTML and Python or Perl−based Scripts.

DTML Tag Syntax

DTML's syntax is similar to HTML. DTML is a tag based mark−up language. In other words DTML uses tags to do its work. Here is a simple snippet of DTML:

```
<dtml−var standard_html_header>

<h1>Hello World!</h1>

<dtml−var standard_html_footer>
```

This DTML code contains two DTML *var* tags and some HTML. The *h1* tags are HTML, not DTML. You typically mix DTML with other mark−up languages like HTML. Normally DTML is used to generate HTML, but there's nothing keeping you from generating other types of text. As you'll see later you can also use DTML to generate mail messages and other textual information.

DTML contains two kinds of tags, *singleton* and *block* tags. Singleton tags consist of one tag enclosed by less−than (<) and greater−than (>) symbols. The *var* tag is an example of a singleton tag:

```
<dtml−var parrot>
```

There's no need to close the *var* tag.

Block tags consist of two tags, one that opens the block and one that closes the block, and content that goes between them:

```
<dtml−in mySequence>

  <!−− this is an HTML comment inside the in tag block −−>

</dtml−in>
```

The opening tag starts the block and the closing tag ends it. The closing tag has the same name as the opening tag with a slash preceding it. This is the same convention that HTML and XML use.

Using DTML Tag Attributes

All DTML tags have attributes. An attribute provides information about how the tag is supposed to work. Some attributes are optional. For example, the *var* tag inserts the value of a variable. It has an optional *missing* attribute that specifies a default value in case the variable can't be found:

```
  <dtml−var wingspan missing="unknown wingspan">
```

If the *wingspan* variable is not found then `unknown wingspan` is inserted instead.

Some attributes don't have values. For example, you can convert an inserted variable to upper case with the *upper* attribute:

```
<dtml−var exclamation upper>
```

Notice that the *upper* attribute, unlike the *missing* attribute doesn't need a value.

Different tags have different attributes. See Appendix A, "DTML Reference", for more information on the syntax of different DTML tags.

Inserting Variables with DTML

Inserting a variable is the most basic task that you can perform with DTML. You already saw how DTML inserts a header and footer into a web page with the *var* tag. Many DTML tags insert variables, and they all do it in a similar way. Let's look more closely at how Zope inserts variables.

Suppose you have a folder whose id is *Feedbags* that has the title "Bob's Fancy Feedbags". Inside the folder create a DTML Method with an id of *pricelist*. Then change the contents of the DTML Method to the following:

```
<dtml−var standard_html_header>

<h1>Price list for <dtml−var title></h1>

<p>Hemp Bag $2.50</p>
<p>Silk Bag $5.00</p>

<dtml−var standard_html_footer>
```

Now view the DTML Method by clicking the *View* tab. You should see an HTML page whose source looks something like this:

```
<html>
<body>

<h1>Price list for Bob's Fancy Feedbags</h1>

<p>Hemp Bag $2.50</p>
<p>Silk Bag $5.00</p>

</body>
</html>
```

This is basically what you might expect. Zope inserts a header, a footer, and a title into the web page. DTML gets the values for these variables from a number of different places. First, the *var* tag tries to find a variable in the current object. Then it looks in the current object's containers. Then it looks in the web request (forms and cookies). If Zope cannot find a variable then it raises an exception, and it stops executing the DTML.

Let's follow this DTML code step by step to see where the variables are found. First Zope looks for *standard_html_header* in the current object, which is the *pricelist* DTML Method. Next, Zope looks for the header in the current object's containers. The *Feedbags* folder doesn't have any methods or properties or sub−objects by that name either. Next Zope examines the *Feedbags* folder's container, and so on until it gets to the root folder. The root folder does have a sub−object named *standard_html_header*. The header object is a DTML Method. So Zope calls the header method and inserts the results.

Next Zope looks for the *title* variable. Here, the search is a little shorter. First, it looks in the *pricelist* DTML Method, which does not have a title, so Zope moves on and finds the *Feedbags* folder's title and inserts it.

Finally Zope looks for the *standard_html_footer* variable. It has to search all the way up to the root folder to find it, just like it looked for *standard_html_header*.

This exercise may seem a bit tedious, but understanding how Zope looks up variables is very important. For example, some important implications of how Zope looks up variables include how Zope objects can get content and behavior from their parents, and how content defined in one location can be reused by many objects.

Processing Input from Forms

It's easy to do form processing with Zope. DTML looks for variables to insert in a number of locations, including information that comes from submitted HTML forms. You don't need any special objects, DTML Documents and DTML Methods will do.

Create two DTML Documents, one with the id *infoForm* and the other with the id *infoAction*. Now edit the contents of the documents. Here's the contents of the *infoForm* document:

```
<dtml-var standard_html_header>

<p>Please send me information on your aardvark adoption
program.</p>

<form action="infoAction">
name: <input type="text" name="user_name"><br>
email: <input type="text" name="email"><br>
<input type="submit">
</form>

<dtml-var standard_html_footer>
```

Now view this document. It is a web form that asks for information and sends it to the *infoAction* document when you submit the form.

Now edit the contents of the *infoAction* document to make it process the form:

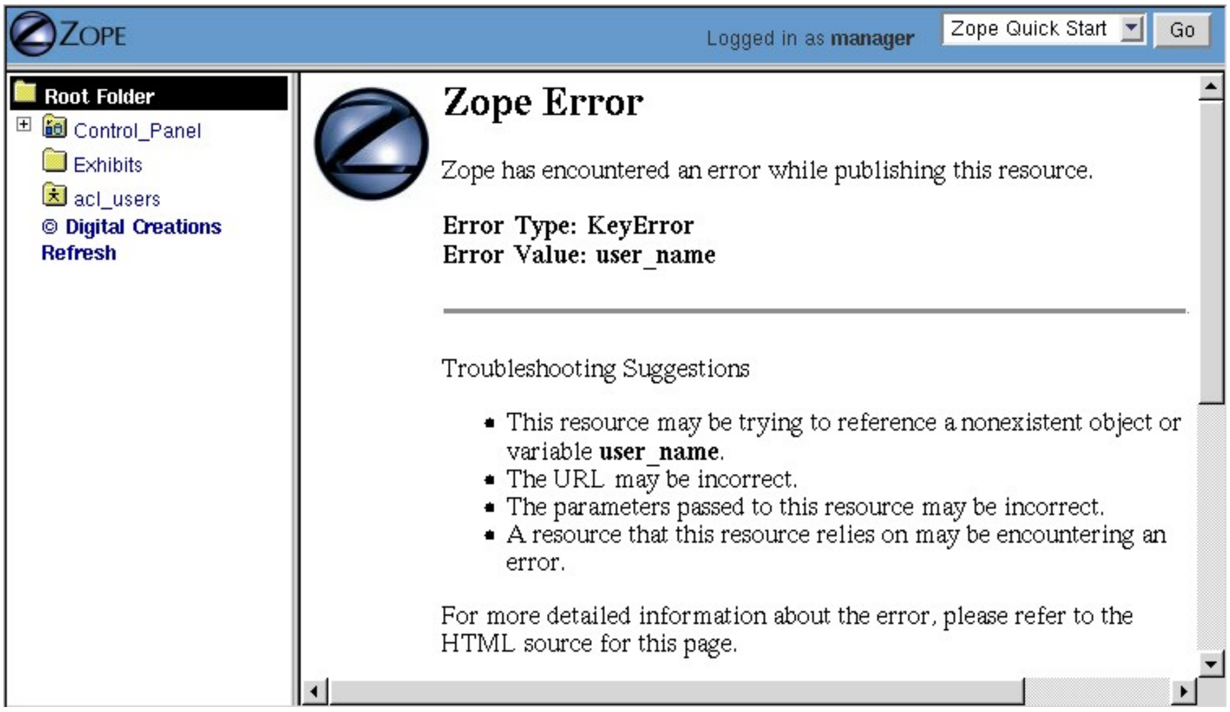
```
<dtml-var standard_html_header>

<h1>Thanks <dtml-var user_name></h1>

<p>We received your request for information and will send you
email at <dtml-var email> describing our aardvark adoption
program as soon as it receives final governmental approval.
</p>

<dtml-var standard_html_footer>
```

This document displays a thanks message which includes name and email information gathered from the web form.

Now go back to the *infoForm* document, view it, fill out the form and submit it. If all goes well you should see a thank you message that includes your name and email address.

The *infoAction* document found the form information from the web request that happened when you clicked the submit button on the *infoForm*. As we mentioned in the last section, DTML looks for variables in a couple of places, one of which is the web request, so there's nothing special you need to do to enable your documents to process web forms.

Let's perform an experiment. What happens if you try to view the *infoAction* document directly, as opposed to getting to it from the *infoForm* document. Click on the *infoAction* document and then click the View tab, as shown in Figure 4−1.



Zope Error

Zope has encountered an error while publishing this resource.

Error Type: KeyError
Error Value: user_name

Troubleshooting Suggestions

- This resource may be trying to reference a nonexistent object or variable **user_name**.
- The URL may be incorrect.
- The parameters passed to this resource may be incorrect.
- A resource that this resource relies on may be encountering an error.

For more detailed information about the error, please refer to the HTML source for this page.

Figure 4−1 DTML error resulting from a failed variable lookup.

Zope couldn't find the *user_name* variable since it was not in the current object, its containers or the web request. This is an error that you're likely to see frequently as you learn Zope. Don't fear, it just means that you've tried to insert a variable that Zope can't find. In this example, you need to either insert a variable that Zope can find, or use the `missing` attribute on the var tag as described above:

```
<h1>Thanks <dtml-var user_name missing="Anonymous User"></h1>
```

Understanding where Zope looks for variables will help you figure out how to fix this kind of problem. In this case, you have viewed a document that needs to be called from an HTML form like *infoForm* in order to provide variables to be inserted in the output.

Dynamically Acquiring Content

Zope looks for DTML variables in the current object's containers if it can't find the variable first in the current object. This behavior allows your objects to find and use content and behavior defined in their parents. Zope uses the term *acquisition* to refer to this dynamic use of content and behavior.

Now that you see how site structure fits into the way names are looked up, you can begin to understand that where you place objects you are looking for is very important.

An example of acquisition that you've already seen is how web pages use standard headers and footers. To acquire the standard header just ask Zope to insert it with the *var* tag:

```
<dtml-var standard_html_header>
```

It doesn't matter where your DTML Method or Document is located. Zope will search upwards until it finds the *standard_html_header* that is defined in the root folder.

You can take advantage of how Zope looks up variables to customize your header in different parts of your site. Just create a new *standard_html_header* in a folder and it will override global header for all web pages in your folder and below it.

Create a folder in the root folder with an id of *Green*. Enter the *Green* folder and create a DTML Document with an id of *welcome*. Edit the *welcome* document to have these contents:

```
<dtml-var standard_html_header>

<p>Welcome</p>

<dtml-var standard_html_footer>
```

Now view the *welcome* document. It should look like a simple web page with the word *welcome*, as shown in Figure 4−2.

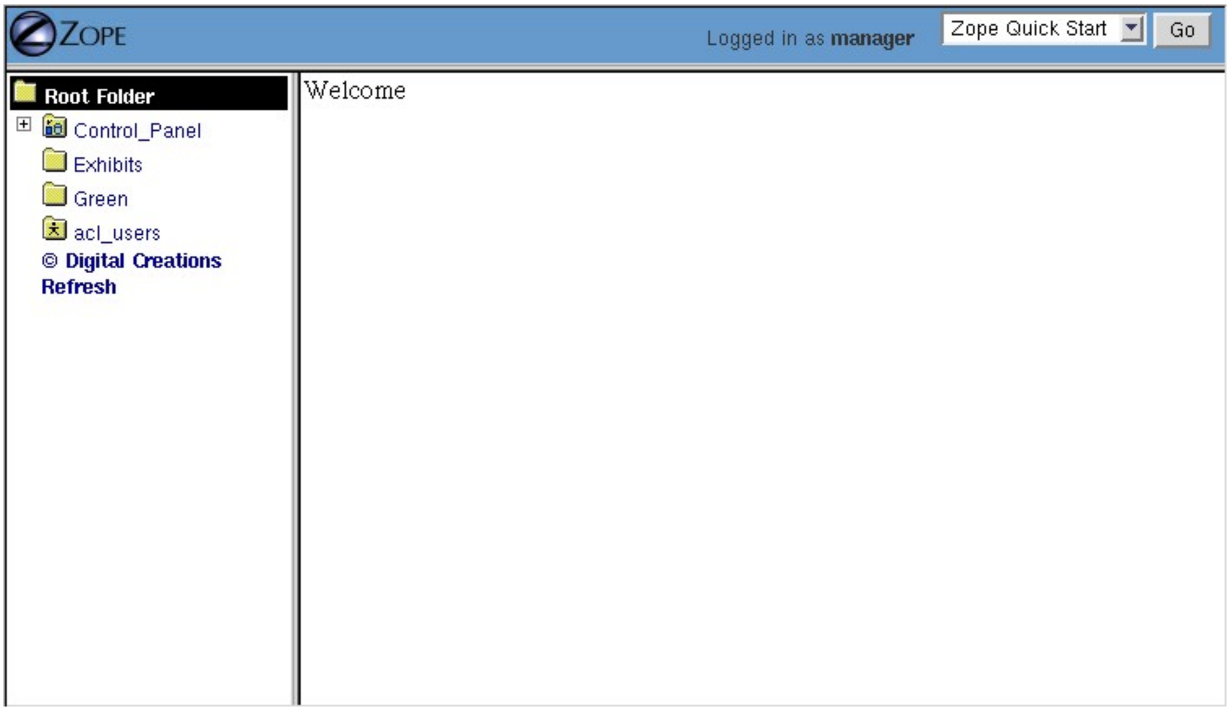


Figure 4−2 Welcome document.

Now let's customize the header for the *Green* folder. Create a DTML Method in the *Green* folder with an id of *standard_html_header*. Then edit the contents of the header to the following:

```
<html>
<head>
  <style type="text/css">
  body {color: #00FF00;}
  p {font-family: sans-serif;}
  </style>
</head>
<body>
```

Notice that this is not a complete web page. This is just a fragment of HTML that will be used as a header. This header uses CSS (Cascading Style Sheets) to make some changes to the look and feel of web pages.

Now go back to the *welcome* document and view it again, as shown in Figure 4−3.

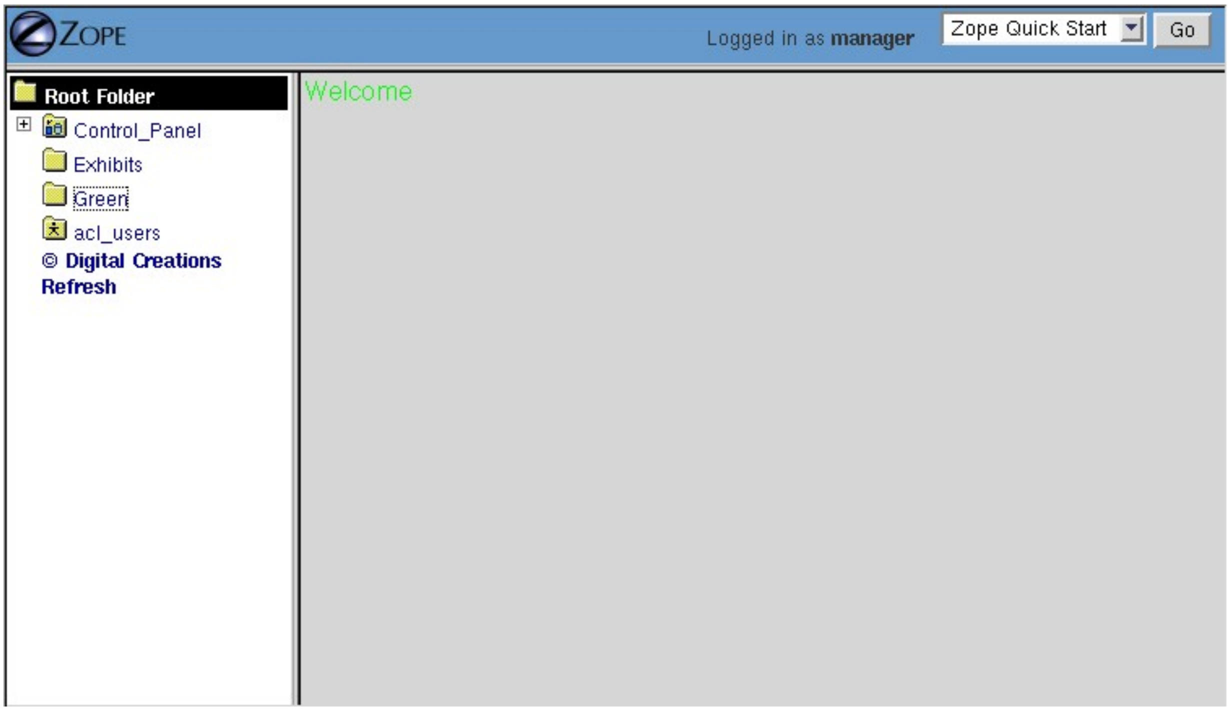


Figure 4−3 Welcome document with custom header.

The document now looks quite different. This is because it is now using the new header we introduced in the *Green* folder. This header will be used by all web pages in the *Green* folder and its sub−folders.

You can continue this process of overriding default content by creating another folder inside the *Green* folder and creating a *standard_html_header* DTML Method there. Now web pages in the sub−folder will use their local header rather than the *Green* folder's header. Using this pattern you can quickly change the look and feel of different parts of your web site. If you later decide that an area of the site needs a different header, just create one. You don't have to change the DTML in any of the web pages; they'll automatically find the closest header and use it.

Using Python Expressions from DTML

So far we've looked at simple DTML tags. Here's an example:

```
<dtml-var getHippo>
```

This will insert the value of the variable named *getHippo*, whatever that may be. DTML will automatically take care of the details, like finding the variable and calling it. We call this basic tag syntax *name* syntax to differentiate it from *expression* syntax.

DTML expressions allow you to be more explicit about how to find and call variables. Expressions are tag attributes that contain snippets of code in the Python language. For example, instead of letting DTML find and call *getHippo*, we can use an expression to explicitly pass arguments:

```
<dtml-var expr="getHippo('with a large net')">
```

Here we've used a Python expression to explicitly call the *getHippo* method with the string argument, `with a large net`. To find out more about Python's syntax, see the Python Tutorial at the Python.org web site. Many DTML tags can use expression attributes.

Expressions make DTML pretty powerful. For example, using Python expressions, you can easily test conditions:

```
<dtml-if expr="foo < bar">
  Foo is less than bar.
</dtml-if>
```

Without expressions, this very simple task would have to be broken out into a separate method and would add a lot of overhead for something this trivial.

Before you get carried away with expressions, take care. Expressions can make your DTML hard to understand. Code that is hard to understand is more likely to contain errors and is harder to maintain. Expressions can also lead to mixing logic in your presentation. If you find yourself staring blankly at an expression for more than five seconds, stop. Rewrite the DTML without the expression and use a Script to do your logic. Just because you can do complex things with DTML doesn't mean you should.

DTML Expression Gotchas

Using Python expressions can be tricky. One common mistake is to confuse expressions with basic tag syntax. For example:

```
<dtml-var objectValues>
```

and:

```
<dtml-var expr="objectValues">
```

will end up giving you two completely different results. The first example of the DTML *var* tag will automatically render variables. In other words it will try to do the right thing to insert your variable, no matter what that variable may be. In general this means that if the variable is a method it will be called with appropriate arguments. This process is covered more thoroughly in Chapter 8, "Variables and Advanced DTML".

In an expression, you have complete control over the variable rendering. In the case of our example, *objectValues* is a method. So:

```
<dtml-var objectValues>
```

will call the method. But:

```
<dtml-var expr="objectValues">
```

will *not* call the method, it will just try to insert it. The result will be not a list of objects but a string such as `<Python Method object at 8681298>`. If you ever see results like this, there is a good chance that you're returning a method, rather than calling it.

To call a method from an expression, you must use standard Python calling syntax by using parenthesis:

```
<dtml-var expr="objectValues()">
```

The lesson is that if you use Python expressions you must know what kind of variable you are inserting and must use the proper Python syntax to appropriately render the variable.

Before we leave the subject of variable expressions we should mention that there is a deprecated form of the expression syntax. You can leave out the "expr=" part on a variable expression tag. But *please* don't do this. It is far too easy to confuse:

```
<dtml-var aName>
```

with:

```
<dtml-var "aName">
```

and get two completely different results. These "shortcuts" were built into DTML long ago, but we do not encourage you to use them now unless you are prepared to accept the confusion and debugging problems that come from this subtle difference in syntax.

The *Var* Tag

The *var* tag inserts variables into DTML Methods and Documents. We've already seen many examples of how the *var* tag can be used to insert strings into web pages.

As you've seen, the *var* tag looks up variables first in the current object, then in its containers and finally in the web request.

The *var* tag can also use Python expressions to provide more control in locating and calling variables.

Var Tag Attributes

You can control the behavior of the *var* tag using its attributes. The *var* tag has many attributes that help you in common formatting situations. The attributes are summarized in Appendix A. Here's a sampling of *var* tag attributes.

html_quote
: This attribute causes the inserted values to be HTML quoted. This means that <, > and & are escaped.

missing
: The missing attribute allows you to specify a default value to use in case Zope can't find the variable. For example:

```
<dtml-var bananas missing="We have no bananas">
```

fmt
: The fmt attribute allows you to control the format of the *var* tags output. There are many possible formats which are detailed in Appendix A.
One use of the *fmt* attribute is to format monetary values. For example, create a *float* property in your root folder called *adult_rate*. This property will represent the cost for one adult to visit the Zoo. Give this property the value `2.2`.

You can display this cost in a DTML Document or Method like so:

```
One Adult pass: <dtml-var adult_rate fmt=dollars-and-cents>
```

This will correctly print "$2.20". It will round more precise decimal numbers to the nearest penny.

Var Tag Entity Syntax

Zope provides a shortcut DTML syntax just for the simple *var* tag. Because the *var* tag is a singleton, it can be represented with an *HTML entity* like syntax:

```
&dtml-cockatiel;
```

This is equivalent to:

```
<dtml-var name="cockatiel" html_quote>
```

The main reason to use the entity syntax is to avoid putting DTML tags inside HTML tags. For example, instead of writing:

```
<input type="text" value="<dtml-var name="defaultValue">">
```

You can use the entity syntax to make things more readable for you and your text editor:

```
<input type="text" value="&dtml-defaultValue;">
```

The *var* tag entity syntax is very limited. You can't use Python expressions and some tag attributes with it. See Appendix A for more information on *var* tag entity syntax.

The *If* Tag

One of DTML's important benefits is to let you customize your web pages. Often customization means testing conditions are responding appropriately. This *if* tag lets you evaluate a condition and carry out different actions based on the result.

What is a condition? A condition is either a true or false value. In general all objects are considered true unless they are 0, None, an empty sequence or an empty string.

Here's an example condition:

objectValues
: True if the variable *objectValues* exists and is true. That is to say, when found and rendered *objectValues* is not 0, None, an empty sequence, or an empty string.

As with the *var* tag, you can use both name syntax and expression syntax. Here are some conditions expressed as DTML expressions.

expr="1"
: Always true.

expr="rhino"
: True if the rhino variable is true.

expr="x < 5"
: True if x is less than 5.

expr="objectValues('File')"
: True if calling the *objectValues* method with an argument of *File* returns a true value. This method is explained in more detail in this chapter.

The *if* tag is a block tag. The block inside the *if* tag is executed if the condition is true.

Here's how you might use a variable expression with the *if* tag to test a condition:

```
<p>How many monkeys are there?</p>

<dtml-if expr="monkeys > monkey_limit">
  <p>There are too many monkeys!</p>
</dtml-if>
```

In the above example, if the Python expression `monkeys > monkey_limit` is true then you will see the first and the second paragraphs of HTML. If the condition is false, you will only see the first.

If tags be nested to any depth, for example, you could have:

```
<p>Are there too many blue monkeys?</p>

<dtml-if "monkeys.color == 'blue'">
  <dtml-if expr="monkeys > monkey_limit">
    <p>There are too many blue monkeys!</p>
  </dtml-if>
</dtml-if>
```

Nested if tags work by evaluating the first condition, and if that condition is true, then evaluating the second. In general, DTML *if* tags work very much like Python *if* statements..

Name and Expression Syntax Differences

The name syntax checks for the *existence* of a name, as well as its value. For example:

```
<dtml-if monkey_house>
  <p>There <em>is</em> a monkey house Mom!</p>
</dtml-if>
```

If the *monkey_house* variable does not exist, then this condition is false. If there is a *monkey_house* variable but it is false, then this condition is also false. The condition is only true is there is a *monkey_house* variable and it is not 0, None, an empty sequence or an empty string.

The Python expression syntax does not check for variable existence. This is because the expression must be valid Python. For example:

```
<dtml-if expr="monkey_house">
  <p>There <em>is</em> a monkey house, Mom!</p>
</dtml-if>
```

This will work as expected as long as *monkey_house* exists. If the *monkey_house* variable does not exist, Zope will raise a *KeyError* exception when it tries to find the variable.

Else and *Elif* Tags

The *if* tag only lets you take an action if a condition is true. You may also want to take a different action if the condition is false. This can be done with the DTML *else* tag. The *if* block can also contain an *else* singleton tag. For example:

```
<dtml-if expr="monkeys > monkey_limit">
  <p>There are too many monkeys!</p>
<dtml-else>
  <p>The monkeys are happy!</p>
</dtml-if>
```

The *else* tag splits the *if* tag block into two blocks, the first is executed if the condition is true, the second is executed if the condition is not true.

A *if* tag block can also contain a *elif* singleton tag. The *elif* tag specifies another condition just like an addition *if* tag. This lets you specify multiple conditions in one block:

```
<dtml−if expr="monkeys > monkey_limit">
  <p>There are too many monkeys!</p>
<dtml−elif expr="monkeys < minimum_monkeys">
  <p>There aren't enough monkeys!</p>
<dtml−else>
  <p>There are just enough monkeys.</p>
</dtml−if>
```

An *if* tag block can contain any number of *elif* tags but only one *else* tag. The *else* tag must always come after the *elif* tags. *Elif* tags can test for condition using either the name or expression syntax.

Using Cookies with the *If* Tag

Let's look at a more meaty *if* tag example. Often when you have visitors to your site you want to give them a cookie to identify them with some kind of special value. Cookies are used frequently all over the Internet, and when they are used properly they are quite useful.

Suppose we want to differentiate new visitors from folks who have already been to our site. When a user visits the site we can set a cookie. Then we can test for the cookie when displaying pages. If the user has already been to the site they will have the cookie. If they don't have the cookie yet, it means that they're new.

Suppose we're running a special. First time zoo visitors get in for half price. Here's a DTML fragment that tests for a cookie using the *hasVisitedZoo* variable and displays the price according to whether a user is new or a repeat visitor:

```
<dtml−if hasVisitedZoo>
  <p>Zoo admission <dtml−var adult_rate fmt="dollars−and−cents">.</p>
<dtml−else>
  <b>Zoo admission for first time visitors
       <dtml−var expr="adult_rate/2" fmt="dollars−and−cents"></p>
</dtml−if>
```

This fragment tests for the *hasVisitedZoo* variable. If the user has visited the zoo before it displays the normal price for admission. If the visitor is here for the first time they get in for half−price.

Just for completeness sake, here's an implementation of the *hasVisitedZoo* method as a Python−based Script:

```
## Script(Python) "hasVisitedZoo"
##parameters=REQUEST, RESPONSE
##
"""
Returns true if the user has previously visited
the Zoo. Uses cookies to keep track of zoo visits.
"""
if REQUEST.has_key('zooVisitCookie'):
    return 1
else:
    RESPONSE.setCookie('zooVisitCookie', '1')
    return 0
```

In Chapter 10, "Advanced Zope Scripting" we'll look more closely at how to script business logic with Python and Perl. For now it is sufficient to see that the method looks for a cookie and returns a true or false value depending on whether the cookie is found or not. Notice how Python uses if and else statements just like DTML uses if and *else* tags. DTML's *if* and *else* tags are based on Python's. In fact Python also has an elif statement, just like DTML.

The *In* Tag

The DTML *in* tag iterates over a sequence of objects, carrying out one block of execution for each item in the sequence. In programming, this is often called *iteration*, or *looping*.

The *in* tag is a block tag like the *if* tag. The content of the *in* tag block is executed once for every iteration in the *in* tag loop. For example:

```
<dtml-in todo_list>
  <p><dtml-var description></p>
</dtml-in>
```

This example loops over a list of objects named *todo_list*. For each item, it inserts an HTML paragraph with a description of the to do item.

Iteration is very useful in many web tasks. Consider a site that display houses for sale. Users will search your site for houses that match certain criteria. You will want to format all of those results in a consistent way on the page, therefore, you will need to iterate over each result one at a time and render a similar block of HTML for each result.

In a way, the contents of an *in* tag block is a kind of *template* that is applied once for each item in a sequence.

Iterating over Folder Contents

Here's an example of how to iterate over the contents of a folder. This DTML will loop over all the files in a folder and display a link to each one. This example shows you how to display all the "File" objects in a folder, so in order to run this example you will need to upload some files into Zope as explained in the previous chapter:

```
<dtml-var standard_html_header>
<ul>
<dtml-in expr="objectValues('File')">
  <li><a href="&dtml-absolute_url;"><dtml-var title_or_id></a></li>
</dtml-in>
</ul>
<dtml-var standard_html_footer>
```

This code displayed the following file listing, as shown in Figure 4−4.

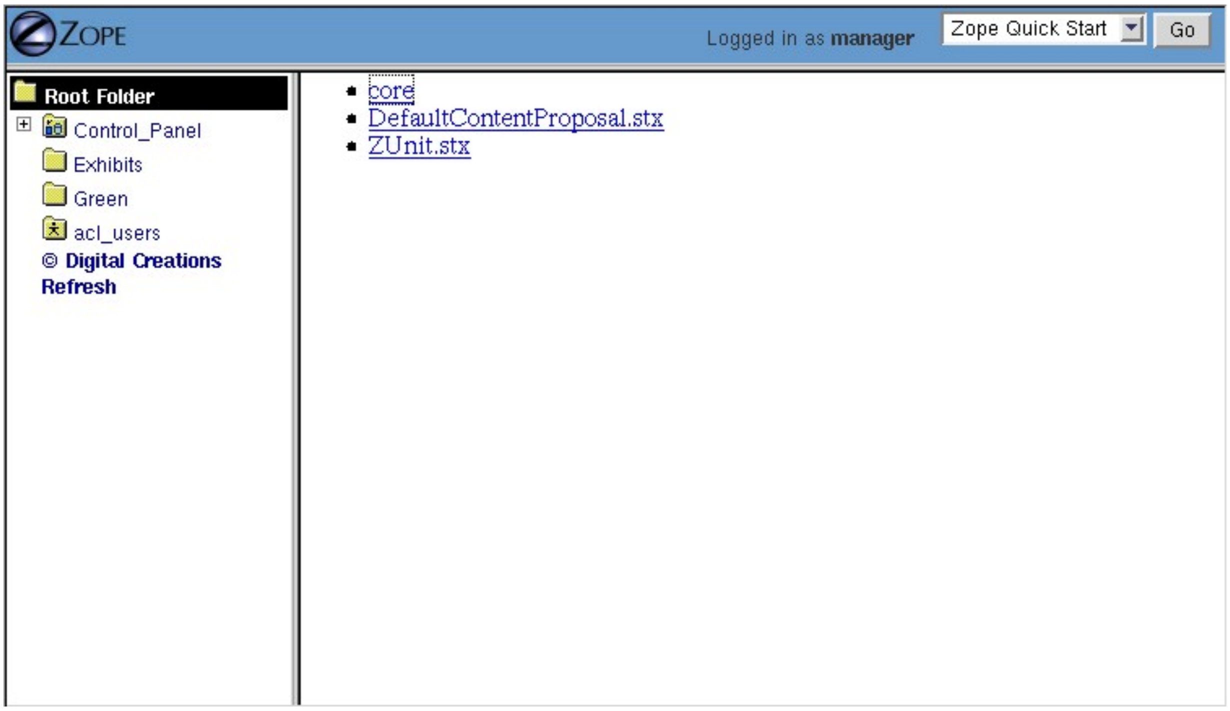


Figure 4−4 Iterating over a list of files.

Let's look at this DTML example step by step. First, the *var* tag is used to insert your common header into the document. Next, to indicate that you want the browser to draw an HTML bulleted list, you have the *ul* HTML tag.

Then there is the *in* tag. The tag has an expression that is calling the Zope API method called *objectValues*. This method returns a sequence of objects in the current folder that match a given criteria. In this case, the objects must be files. This method call will return a list of files in the current folder.

The *in* tag will loop over every item in this sequence. If there are four file objects in the current folder, then the *in* tag will execute the code in its block four times; once for each object in the sequence.

During each iteration, the *in* tag looks for variables in the current object, first. In Chapter 8, "Variables and Advanced DTML" we'll look more closely at how DTML looks up variables.

For example, this *in* tag iterates over a collection of File objects and uses the *var* tag to look up variables in each file:

```
<dtml-in expr="objectValues('File')">
  <li><a href="&dtml-absolute_url;"><dtml-var title_or_id></a></li>
</dtml-in>
```

The first *var* tag is an entity and the second is a normal DTML *var* tag. When the *in* tag loops over the first object its *absolute_url* and *title_or_id* variables will be inserted in the first bulleted list item:

```
<ul>
  <li><a href="http://localhost:8080/FirstFile">FirstFile</a></li>
```

During the second iteration the second object's *absolute_url* and *title_or_id* variables are inserted in the output:

```
<ul>
```

```
        <li><a href="http://localhost:8080/FirstFile">FirstFile</a></li>
        <li><a href="http://localhost:8080/SecondFile">SecondFile</a></li>
```

This process will continue until the *in* tag has iterated over every file in the current folder. After the *in* tag you finally close your HTML bulleted list with a closing *ul* HTML tag and the *standard_html_footer* is inserted to close the document.

In Tag Special Variables

The *in* tag provides you with some useful information that lets you customize your HTML while you are iterating over a sequence. For example, you can make your file library easier to read by putting it in an HTML table and making every other table row an alternating color, like this, as shown in Figure 4−5.

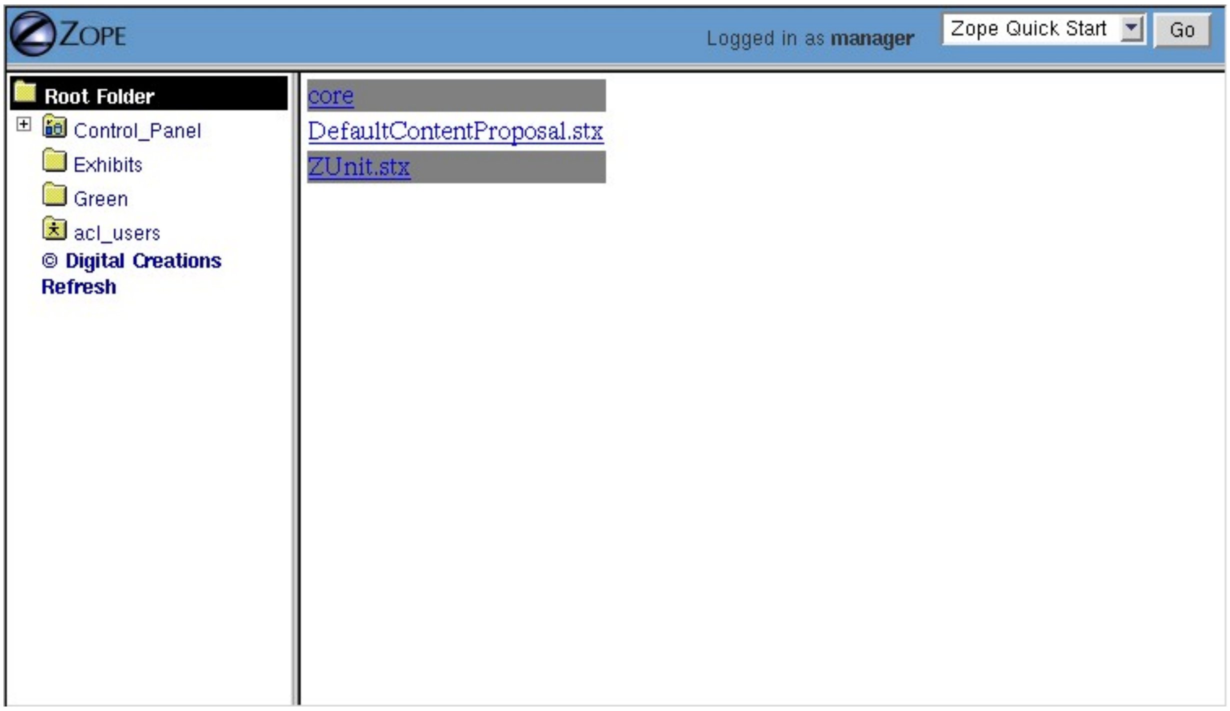


Figure 4−5 File listing with alternating row colors.

The *in* tag makes this easy. Change your file library method a bit to look like this:

```
<dtml-var standard_html_header>

<table>
<dtml-in expr="objectValues('File')">
  <dtml-if sequence-even>
    <tr bgcolor="grey">
  <dtml-else>
    <tr>
  </dtml-if>
  <td>
  <a href="&dtml-absolute_url;"><dtml-var title_or_id></a>
  </td></tr>
</dtml-in>
</table>

<dtml-var standard_html_footer>
```

Here an *if* tag is used to test for a special variable called `sequence−even`. The *in* tag sets this variable to a true or false value each time through the loop. If the current iteration number is even, then the value is true, if the iteration number is odd, it is false.

The result of this test is that a *tr* tag with either a gray background or no background is inserted into the document for every other object in the sequence. As you might expect, there is a `sequence−odd` that always has the opposite value of `sequence−even`.

There are many special variables that the *in* tag defines for you. Here are the most common and useful:

sequence−item
: This special variable is the current item in the iteration.
: In the case of the file library example, each time through the loop the current file of the iteration is assigned to sequence−item. It is often useful to have a reference to the current object in the iteration.

sequence−index
: the current number, starting from 0, of iterations completed so far. If this number is even, `sequence−even` is true and `sequence−odd` is false.

sequence−number
: The current number, starting from 1, of iterations completed so far. This can be thought of as the cardinal position (first, second, third, etc.) of the current object in the loop. If this number is even, `sequence−even` is false and `sequence−odd` is true.

sequence−start
: This variable is true for the very first iteration.

sequence−end
: This variable is true for the very last iteration.

These special variables are detailed more thoroughly in Appendix A.

DTML is a powerful tool for creating dynamic content. It allows you to perform fairly complex calculations. In Chapter 8, "Variables and Advanced DTML", you'll find out about many more DTML tags, and more powerful ways to use the tags you already have seen. Despite its power, you should resist the temptation to use DTML for complex scripting. In Chapter 10, "Advanced Zope Scripting" you'll find out about how to use Python and Perl for scripting business logic.

Chapter 5. Using Zope Page Templates

Page Templates are a web page generation tool. They help programmers and designers collaborate in producing dynamic web pages for Zope web applications. Designers can use them to maintain pages without having to abandon their tools, while preserving the work required to embed those pages in an application. In this chapter, you'll learn the basics about Page Templates including how you can use them in your web site to create dynamic web pages easily. In Chapter 9, "Advanced Page Templates", you'll learn about advanced Page Template features.

The goal of Page Templates is to allow designers and programmers to work together easily. A designer can use a WYSIWYG HTML editor to create a template, then a programmer can edit it to make it part of an application. If required, the designer can load the template *back* into his editor and make further changes to its structure and appearance. By taking reasonable steps to preserve the changes made by the programmer, the designer will not disrupt the application.

Page Templates aim at this goal by adopting three principles:

1. Play nicely with editing tools.
2. What you see is very similar to what you get.
3. Keep code out of templates, except for structural logic.

A Page Template is like a model of the pages that it will generate. In particular, it is a valid HTML page.

Zope Page Templates versus DTML

Zope already has DTML, why do you need another template language. First of all, DTML is not aimed at HTML designers. Once a page has been converted into a template, it is invalid HTML, making it difficult to work with outside of the application. Secondly, DTML suffers from a failure to separate presentation, logic, and content (data). This decreases the scalability of content management and website development efforts that use these systems.

DTML can do things that Page Templates can't such as dynamically generate email messages (Page Templates can only generate HTML and XML). So DTML won't go away. However, we do see Page Templates taking over almost all HTML/XML presentation work in Zope.

How Page Templates Work

Page Templates use the Template Attribute Language (TAL). TAL consists of special tag attributes. For example, a dynamic page title might look like this:

```
<title tal:content="here/title">Page Title</title>
```

The `tal:content` attribute is a TAL statement. Since it has an XML namespace (the `tal:` part) most editing tools will not complain that they don't understand it, and will not remove it. It will not change the structure or appearance of the template when loaded into a WYSIWYG editor or a web browser. The name `content` indicates that it will set the content of the `title` tag, and the value "here/title" is an expression providing the text to insert into the tag.

All TAL statements consist of tag attributes whose name starts with `tal:` and all TAL statements have values associated with them. The value of a TAL statement is shown inside quotes. See Appendix C, "Zope Page Templates Reference", for more information on TAL.

To the HTML designer using a WYSIWYG tool, the dynamic title example is perfectly valid HTML, and shows up in their editor looking like a title should look like. In other words, Page Templates play nicely with editing tools.

This example also demonstrates the principle that, "What you see is very similar to what you get". When you view the template in an editor, the title text will act as a placeholder for the dynamic title text. The template provides an example of how generated documents will look.

When this template is saved in Zope and viewed by a user, Zope turns the dummy content into dynamic content, replacing "Page Title" with whatever "here/title" resolves to. In this case, "here/title" resolves to the title of the object to which to the template is applied. This substitution is done dynamically, when the template is viewed.

There are template statements for replacing entire tags, their contents, or just some of their attributes. You can repeat a tag several times or omit it entirely. You can join parts of several templates together, and specify simple error handling. All of these capabilities are used to generate document structures. Despite these capabilities, you ***can't*** create subroutines or classes, perform complex flow control, or easily express complex algorithms. For these tasks, you should use Python−based Scripts or application components.

The Page Template language is deliberately not as powerful and general−purpose as it could be. It is meant to be used inside of a framework (such as Zope) in which other objects handle business logic and tasks unrelated to page layout.

For instance, template language would be useful for rendering an invoice page, generating one row for each line item, and inserting the description, quantity, price, and so on into the text for each row. It would not be used to create the invoice record in a database or to interact with a credit card processing facility.

Creating a Page Template

If you design pages, you will probably use FTP or WebDAV instead of the Zope Management Interface (ZMI) to create and edit Page Templates. See the "Using FTP and WebDAV" section later in this chapter for information on editing Page Templates remotely. For the small examples in this chapter, it is easier to use the ZMI.

Use your web browser to log into the Zope Management Interface as a manager. Choose a Folder to work in (the root is fine) and pick "Page Template" from the drop−down add list. Type "simple_page" in the add form's *Id* field, then push the "Add and Edit" button.

You should now see the main editing page for the new Page Template. The title is blank, the content−type is `text/html`, and the default template text is in the editing area.

Now let's create simple dynamic page. Type the words "a Simple Page" in the *Title* field. Then, edit the template text to look like this:

```
<html>
 <body>
   <p>
     This is <b tal:replace="template/title">the Title</b>.
   </p>
 </body>
</html>
```

Now push the *Save Changes* button. Zope should show a message confirming that your changes have been saved.

If an HTML comment starting with `<-- Page Template Diagnostics` is added to the template text, then check to make sure you typed the example correctly and save it again. This comment is an error message telling you that something is wrong. You don't need to erase the error comment; once the error is corrected it will go away.

Click on the *Test* tab. You should see a page with, "This is a Simple Page." at the top. Notice that the text is plain; nothing is in bold.

Back up, then click on the *Browse HTML source* link under the content−type field. This will show you the *unrendered* source of the template. You should see, "This is ***the Title***." Back up again, so that you are ready to edit the example further.

The *Content−Type* field allows you to specify the content type of your page. Generally you'll use a content type of `text/html` HTML or `text/xml` for XML.

If you set the content−type to `text/html` then Zope parses your template using HTML compatiblity mode which allowers HTML's loose markup. If you set your content−type to something other than `text/html` then Zope assumes that your template is well formed XML. Zope also requires an explicit TAL and METAL XML namespace declarations for well formed XML.

The *Expand macros with editing* control is explain in Chapter 9, "Advanced Page Templates".

Simple Expressions

The expression, "template/title" in your simple Page Template is a *path expression*. This the most common type of expression. There are several other types of expressions defined by the TAL Expression Syntax (TALES) standard. For more information on TALES see Appendix C, "Zope Page Templates Reference".

The "template/title" path expression fetches the `title` property of the template. Here are some other common path expressions:

- `request/URL`: The URL of the current web request.
- `user/getUserName`: The authenticated user's login name.
- `container/objectIds`: A list of Ids of the objects in the same Folder as the template.

Every path starts with a variable name. If the variable contains the value you want, you stop there. Otherwise, you add a slash ('/') and the name of a sub−object or property. You may need to work your way through several sub−objects to get to the value you're looking for.

Zope defines a small set of built−in variables such as `request` and `user`, which are described in Chapter 9, "Advanced Page Templates". You will also learn how to define your own variables in that chapter.

Inserting Text

In your "simple_page" template, you used the `tal:replace` statement on a bold tag. When you tested it, Zope replaced the entire tag with the title of the template. When you browsed the source, you saw the template text in bold. We used a bold tag in order to highlight the difference.

In order to place dynamic text inside of other text, you typically use `tal:replace` on a `span` tag rather than on a bold tag. For example, add the following lines to your example:

```
<br>
The URL is <span tal:replace="request/URL">URL</span>.
```

The `span` tag is structural, not visual, so this looks like "The URL is URL." when you view the source in an editor or browser. When you view the rendered version, it may look something like:

```
<br>
The URL is http://localhost:8080/simple_page.
```

If you want to insert text into a tag but leave the tag itself alone, you use the `tal:content` statement. To set the title of your example page to the template's title property, add the following lines between the `html` and the `body` tags:

```
<head>
  <title tal:content="template/title">The Title</title>
</head>
```

If you open the "Test" tab in a new browser window, the window's title will be "a Simple Page". If you view the source of the page you'll see something like this:

```
<html>
  <head>
    <title>a Simple Page</title>
  </head>
...
```

Zope inserted the title of your template into the `title` tag.

Repeating Structures

Now let's add some context to your page, in the form of a list of the objects that are in the same Folder as the template. You will make a table that has a numbered row for each object, and columns for the id, meta−type, and title. Add these lines to the bottom of your example template:

```
<table border="1" width="100%">
  <tr>
    <th>Number</th>
    <th>Id</th>
    <th>Meta−Type</th>
    <th>Title</th>
  </tr>
  <tr tal:repeat="item container/objectValues">
    <td tal:content="repeat/item/number">#</td>
    <td tal:content="item/getId">Id</td>
    <td tal:content="item/meta_type">Meta−Type</td>
    <td tal:content="item/title">Title</td>
  </tr>
</table>
```

The `tal:repeat` statement on the table row means "repeat this row for each item in my container's list of object values". The repeat statement puts the objects from the list into the `item` variable one at a time (this is called the *repeat variable*), and makes a copy of the row using that variable. The value of "item/getId" in each row is the Id of the object for that row, and likewise with "item/meta_type" and "item/title".

You can use any name you like for the repeat variable ("item" is only an example), as long as it starts with a letter and contains only letters, numbers, and underscores ('_'). The repeat variable is only defined in the repeat tag. If you try to use it above or below the `tr` tag you will get an error.

You can also use the repeat variable name to get information about the current repetition. By placing it after the built−in variable `repeat` in a path, you can access the repetition count from zero ('index'), from one ('number'), from "A" ('Letter'), and in several other ways. So, the expression `repeat/item/number` is `1` in the first row, `2` in the second row, and so on.

Since one `tal:repeat` loop can be placed inside of another, more than one can be active at the same time. This is why you must write `repeat/item/number` instead of just `repeat/number`. You must specify which loop your interested in by including the loop name.

Now view the page and notice how it lists all the objects in the same folder as the template. Try adding or deleting objects from the folder and notice how the page reflects these changes.

Conditional Elements

Using Page Templates you can dynamically query your environment and selectively insert text depending on conditions. For example, you could display special information in response to a cookie:

```
<p tal:condition="request/cookies/verbose | nothing">
  Here's the extra information you requested.
</p>
```

This paragraph will be included in the output only if there is a `verbose` cookie set. The expression, 'request/cookies/verbose | nothing' is true only when there is a cookie named `verbose` set. You'll learn more about this kind of expression in Chapter 9, "Advanced Page Templates".

Using the `tal:condition` statement you can check all kinds of conditions. A `tal:condition` statement does nothing if its expression has a true value, but removes the entire statement tag, including its contents, if the value is false. Zope considers the number zero, a blank string, an empty list, and the built−in variable `nothing` to be false values. Nearly every other value is true, including non−zero numbers, and strings with anything in them (even spaces!).

Another common use of conditions is to test a sequence to see if it is empty before looping over it. For example is the last section you saw how to draw a table by iterating over a collection of objects. Here's how to add a check to page so that if the list of objects is empty no table is drawn:

```
<table tal:condition="container/objectValues"
       border="1" width="100%">
  <tr>
    <th>Number</th>
    <th>Id</th>
    <th>Meta−Type</th>
    <th>Title</th>
  </tr>
  <tr tal:repeat="item container/objectValues">
    <td tal:content="repeat/item/number">#</td>
    <td tal:content="item/getId">Id</td>
    <td tal:content="item/meta_type">Meta−Type</td>
    <td tal:content="item/title">Title</td>
  </tr>
</table>
```

If the expressions, `container/objectValues` is false then the entire table is omitted.

Changing Attributes

Most, if not all, of the objects listed by your template have an `icon` property, that contains the path to the icon for that kind of object. In order to show this icon in the meta−type column, you will need to insert this path into the `src` attribute of an `img` tag. Edit the meta−type column in both rows to look like this:

```
<td><img src="/misc_/OFSP/Folder_icon.gif"
         tal:attributes="src item/icon">
    <span tal:replace="item/meta_type">Meta−Type</span>
</td>
```

The `tal:attributes` statement replaces the `src` attribute of the `img` tag with the value of `item/icon`. The `src="/misc_/OFSP/Folder_icon.gif"` attribute in the template acts as a placeholder.

Notice that we've replaced the `tal:content` attribute on the table cell with a tal:replace statement on a `span` tag. This change allows you to have both an image and text in the table cell.

Creating a File Library with Page Templates

Here's an example of using Page Templates with Zope to create a simple file library with one template, a little bit of Python code, and some files.

First, create a mock up of a file library page using your HTML editor. The examples in this chapter were made with Amaya. This mock−up doesn't need to overdo it, it just shows some dummy information. Here's a mock−up of a file library that contains one file:

```
<!DOCTYPE html PUBLIC "−//W3C//DTD HTML 4.01 Transitional//EN"
                      "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
  <title>File Library</title>
  <style type="text/css">
  .header {
    font−weight: bold;
    font−family: helvetica;
    background: #DDDDDD;
  }
  h1 {
    font−family: helvetica;
  }
  .filename {
    font−family: courier
  }
  </style>
  <meta name="GENERATOR" content="amaya 5.1">
</head>

<body>
<h1>File Library</h1>

<p>Click on a file below to download it.</p>

<table border="1" cellpadding="5" cellspacing="0">
  <tbody>
    <tr>
      <td class="header">Name</td>
      <td class="header">Type</td>
      <td class="header">Size</td>
```

```
        <td class="header">Last Modified</td>
      </tr>
      <tr>
        <td><a href="Sample.tgz" class="filename">Sample.tgz</a></td>
        <td>application/x-gzip-compressed</td>
        <td>22 K</td>
        <td>2001/09/17</td>
      </tr>
    </tbody>
  </table>
  </body>
  </html>
```

Now, log into your Zope and create a folder called `FileLib`. In this folder, create a Page Template called `index_html` by selecting `Page Template` from the add menu, specifying the Id `index_html` in the form, and clicking *Add*.

Now, with your HTML editor, save the above HTML to the URL of the `index_html` Page Template followed by `/source.html`, for example, `http://localhost:8080/FileLib/index_html/source.html`. Notice that the URL to *save* the `index_html` page ends in `source.html`. Because Page Templates are dynamic, you need a way to edit the raw source of the template, unrendered by the page template language. Appending `source.html` to a Page Template gives you this raw source. Note, if the content−type of your page is `text/xml` then you'll use `source.xml`, rather than `source.html`.

Now that you've saved the template, you can go back to Zope and click on `index_html` and then click on its *Test* tab to view the template. It looks just like it the mock−up, so everything is going well.

Now let's tweak the above HTML and add some dynamic magic. First, we want the title of the template to be dynamic. In Zope, you'll notice that the Page Template has a *title* form field that you can fill in. Instead of being static HTML, we want Zope to dynamically insert the Page Templates title into the rendered version of the template. Here's how:

```
<head>
  ...
  <title tal:content="template/title">File Library</title>
  ...

<body>
<h1 tal:content="template/title">File Library</h1>
...
```

Now go to Zope and change the title of the `index_html` page template. After saving that change, click the *Test* tab. As you can see, the Page Template dynamically inserted the title of the template object in the output of the template.

Notice the new `content` tag attribute in the `tal` xml namespace. This attribute says to "replace the *content* of this tag with the variable 'template/title'". In this case, `template/title` is the title of the `index_html` Page Template.

The next bit of magic is to build a dynamic file list that shows you all the File objects in the `FileLib` folder.

To start, you need to write just one line of Python. Go to the `FileLib` folder and create a `Script (Python)` in that folder. Give the script the id `files` and click *Add and Edit*. Edit the script to contain the following Python code:

```
## Script (Python) "files"
##
return container.objectValues('File')
```

This will return a list of any File objects in the FileLib folder. Now, edit your `index_html` Page Template and add some more `tal` attributes to your mock−up:

```
...
<tr tal:repeat="item container/files">
  <td><a href="Sample.tgz" class="filename"
         tal:attributes="href item/getId"
         tal:content="item/getId">Sample.tgz</a></td>
  <td tal:content="item/content_type">application/x−gzip−compressed</td>
  <td tal:content="item/getSize">22 K</td>
  <td tal:content="item/bobobase_modification_time">2001/09/17</td>
</tr>
...
```

The interesting part is the `tal:repeat` attribute on the `tr` HTML tag. This attribute tells the template to iterate over the values returned by "container/files" (the Python script you created) and create a new table row for each of those files. During each iteration, the current file object being iterated over is assigned the name `item`.

The cells of each row all have `tal:content` attributes that describe the the data that should go in each cell. During each iteration through the table row loop, the id, the content type, the size, and modification times replace the dummy data in the rows. Also notice how the anchor link dynamically points to the current file using `tal:attributes` to rewrite the `href` attribute.

This data comes from the `item` object by calling Zope API methods on what we know is a file object. The methods `item/getId`, `item/content_type`, `item/getSize`, `item/bobobase_modification_time` are all standard API functions that are documented in Zope's online help system.

Go to Zope and test this script by first uploading some Files into the `FileLib` folder. This is done by selecting `File` from the add menu and clicking on the `upload` form button on the next screen. After uploading your file, you can just click *Add*. If you do not specify an id, then the filename of the file you are uploading will be used.

After uploading some files, go to the `index_html` Page Template and click the *Test* tab. Now, you can see the Page Template has rendered a very simple file library with just a few HTML tag attribute changes.

There are a few cosmetic problems with the file library as it stands. The size and date displays are not very pretty and don't match the format of the dummy content. You would like the size of the files to be displayed in K or MB rather than bytes. Here's a Python−based script that you can use for this:

```
## Script (Python) "file_size"
##
"""
Return a string describing the size of a file.
"""
bytes=context.getSize()
k=bytes/1024.0
mb=bytes/1048576.0
if mb > 1:
    return "%.2f MB" % mb
if k > 1:
    return "%d K" % k
```

```
      return "%d bytes" % bytes
```

Create this script with the Id `file_size` in your `FileLib` folder. It calculates a file's size in kilobytes and megabytes and returns an appropriate string describing the size of the file. Now you can use the script in place of the `item/getSize` expression:

```
...
<td tal:content="item/file_size">22 K</td>
...
```

You can also fix the date formatting problems with a little Python. Create a script named `file_date` in your `FileLib` folder:

```
## Script (Python) "file_date"
##
"""
Return modification date as string YYYY/MM/DD
"""
date=context.bobobase_modification_time()
return "%s/%s/%s" % (date.year(), date.mm(), date.day())
```

Now replace the `item/bobobase_modification_time` expression with a reference to this script:

```
...
<td tal:content="item/file_date">2001/09/17</td>
...
```

Congratulations, you've successfully taken a mock−up and turned it into a dynamic Page Template. This example illustrates how Page Templates work well as the "presentation layer" to your applications. The Page Templates present the application logic (the Python−based scripts) and the application logic works with the data in your site (the files).

Remote Editing with FTP and WebDAV

You can edit Page Templates remotely with FTP and WebDAV, as well as HTTP PUT publishing. Using these methods, you can use Page Templates without leaving advanced WYSIWYG editors such as DreamWeaver.

The previous section showed you how to edit a page remotely using Amaya, which uses HTTP PUT to upload pages. You can do the same thing with FTP and WebDAV using the same steps.

1. Create a Page Template in the Zope Management interface. You can name it with whatever file extension you wish. Many folks prefer `.html`, while others prefer `.zpt`. Note, some names such as `index_html` have special meanings to Zope.
2. Retrieve the file using the URL of you page template plus `/source.html` or `/source.xml`. This gives you the source of your Page Template.
3. Edit your file with your editor and then save it. When you save it you should use the same source URL you used to retrieve it.
4. Optionally reload your page after you edit it, to check for error comments. See the next section for more details on debugging.

In later versions of Zope you'll probably be able to create Page Templates without using the Zope Management Interface.

Debugging and Testing

Zope helps you find and correct problems in your Page Templates. Zope notices problem at two different times: when you're editing a Page Template, and when you're viewing a Page Template. Zope catches different types of problems when you're editing than when you're viewing a Page Template.

You're probably already familiar with trouble−shooting comments that Zope inserts into your Page Templates when it runs into problems. These comments tell you about problems that Zope finds while you're editing your templates. The sorts of problems that Zope finds when you're editing are mostly errors in your `tal` statements. For example:

```
<!−− Page Template Diagnostics
 Compilation failed
 TAL.TALDefs.TALError: bad TAL attribute: 'contents', at line 10, column 1
−−>
```

This diagnostic message lets you know that you mistakenly used `tal:contents` rather than `tal:content` on line 10 of your template. Other diagnostic messages will tell you about problems with your template expressions and macros.

When you're using the Zope management interface to edit Page Templates it's easy to spot these diagnostic messages. However, if you're using WebDAV or FTP it's easy to miss these messages. For example, if you save a template to Zope with FTP, you won't get an FTP error telling you about the problem. In fact, you'll have to reload the template from Zope to see the diagnostic message. When using FTP and WebDAV it's a good idea to reload templates after you edit them to make sure that they don't contain diagnostic messages.

If you don't notice the diagnostic message and try to render a template with problems you'll see a message like this:

```
Error Type: RuntimeError
Error Value: Page Template hello.html has errors.
```

That's your signal to reload the template and check out the diagnostic message.

In addition to diagnostic messages when editing, you'll occasionally get regular Zope errors when viewing a Page Template. These problems are usually due to problems in your template expressions. For example, you might get an error if an expression can't locate a variable:

```
Error Type: Undefined
Error Value: "unicorn" not found in "here/unicorn"
```

This error message tells you that it cannot find the `unicorn` variable which is referenced in the expression, `here/unicorn`. To help you figure out what went wrong, Zope includes information about the environment in the traceback. If you're in debugging mode this information will be available at the bottom of the error page. Otherwise, view the source of the error page to see the traceback. The traceback will include information about the environment:

```
...
'here': <Application instance at 01736F78>,
'modules': <Products.PageTemplates.ZRPythonExpr._SecureModuleImporter instance at 016E77FC>
'nothing': None,
'options': {'args': ()},
'request': ...
'root': <Application instance at 01736F78>,
'template': <ZopePageTemplate instance at 01732978>,
'traverse_subpath': [],
'user': amos})
...
```

This information is a bit cryptic, but with a little detective work it can help you figure out what went wrong. In this case, it tells us that the `here` variable is an "Application instance". This means that it is the top−level Zope folder (notice how `root` variable is the same "Application instance"). Perhaps the problem is that you wanted to apply the template to a folder that had a `unicorn` property. The traceback doesn't provide a lot of help, but it can help you sometimes.

XML Templates

Another example of the flexibility of Page Templates is that they can dynamically render XML as well as HTML. For example, in Chapter 5, "Creating Basic Zope Applications", you created the following XML:

```
<guestbook>
  <entry>
    <comments>My comments</comments>
  </entry>
  <entry>
    <comments>I like your web page</comments>
  </entry>
  <entry>
    <comments>Please no blink tags</comments>
  </entry>
</guestbook>
```

This XML was created by looping over all the DTML Documents in a folder and inserting their source into `comment` elements. In this section, we'll show you how to use Page Templates to generate this same XML.

Create a new Page Template called "entries.xml" in your guest book folder with the following contents:

```
<guestbook xmlns:tal="http://xml.zope.org/namespaces/tal">
  <entry tal:repeat="entry python:here.objectValues('DTML Document')">
    <comments tal:content="entry/document_src">Comment goes here...</comments>
  </entry>
</guestbook>
```

Make sure you set the content type to `text/xml`. Now, click *Save Changes* and click the *Test* tab. If you're using Netscape, it will prompt you to download an XML document, if you are using MSIE 5 or higher, you will be able to view the XML document in the browser.

Notice how the `tal:repeat` statement loops over all the DTML Documents. The `tal:content` statement inserts the source of each document into the `comments` element. The `xmlns:tal` attribute is an XML namespace declaration. It tells Zope that names that start with `tal` are Page Template commands. See Appendix C, "Zope Page Templates Reference" for more information about TAL and TALES XML namespaces.

Creating XML with Page Templates is almost exactly like creating HTML. The most important difference is that you must use explicit XML namespace declarations. Another difference is that you should set the content type to `text/xml` or whatever the content−type for your XML should be. The final difference is that you can browse the source of an XML template by going to `source.xml` rather than `source.html`.

Using Templates with Content

In general Zope supports content, presentation, and logic components. Page Templates are presentation components and they can be used to display content components.

Zope 2.5 ships with several content components: ZSQL Methods, Files, and Images. DTML Documents and methods are not really pure content components since they can hold content and execute DTML code. As this time Zope doesn't come with a good general purpose content object. You can use Files for textual content since you can edit the contents of Files if the file is less than 64K and contains text. However, the File object is pretty basic.

Zope's Content Management Framework (CMF) solves this problem by providing an assortment of rich content components. The CMF is Zope's content management add on. It introduces all kinds of enhancements including work−flow, skins, and content objects. The CMF makes a lot of use of Page Templates. A later release of Zope will probably include the CMF.

Conclusion

Zope Page Templates help you build web pages for your web applications. Templates make it easier for you to use normal HTML tools and techniques to build web pages. They also provide convenient hooks to allow you to attach them to your applications. Page Templates help designers and programmers work together to produce web applications. In Chapter 9, "Advanced Page Templates", you'll learn about powerful template techniques like Python expressions, and macros.

Chapter 6: Creating Basic Zope Applications

In Chapter 3, "Using Basic Zope Objects" and Chapter 4, "Dynamic Content with DTML" you learned about basic Zope objects and DTML. In this chapter you'll see how you can build simple but powerful web applications using these tools. In later chapters of the book you'll discover more complex objects and more complex DTML. However, the design techniques covered in this chapter are still relevant.

Note: in chapter 3, "Basic Zope Objects", we explained how Zope Page Templates are new to Zope and should be used for presentation. We have not yet converted this chapter over to use Page Templates instead of DTML. We will be rewriting this chapter soon to reflect new methedologies based on page templates soon.

Building Applications with Folders

Folders are the "basic building blocks" of Zope applications. Folders allow you to organize your Zope objects, and actively participate in your web applications. Folders are given behavior by adding scripts to them.

Scripts and folders work together to build simple applications. Folders provide structure for your information and also provide a framework for your site's behavior. Later in this chapter, an example of a simple guest book application based on this design concept is given. A folder is used to hold the methods, scripts and data of the guest book application, the scripts provide behavior that define how the application works, and the methods provide presentation to the application.

For example, suppose you have an *Invoices* folder to hold invoices. You could create objects inside that folder named *addInvoice* and *editInvoice* to allow you to add and edit invoices. Now your *Invoices* folder becomes a small application.

Zope's simple and expressive URLs are used to work with the invoices application. As you've seen, you can display a Zope object by going to its URL in your browser. So for example, the URL *http://localhost:8080/Invoices/addInvoice* calls the *addInvoice* object on the *Invoices* folder. This URL might take you to a screen that lets you add an invoice. Likewise, the URL *http://localhost:8080/Invoices/editInvoice?invoice_number=42* calls the *editInvoice* object on the *Invoices* folder and passes it the argument *invoice_number* with a value of 42. This URL could allow you to edit invoice number 42.

Calling Objects on Folders with URLs

The invoices example demonstrates a powerful Zope feature. You can call an object on a folder by going to a URL that consists of the folder's URL followed by the id of the object. This facility is used throughout Zope and is a very general design pattern. In fact you are not just restricted to calling objects on folders. You'll see later how you can call objects on all kinds of Zope objects using the same URL technique.

For example suppose you want to call an object named *viewFolder* on one of your folders. Perhaps you have many different *viewFolder* objects in different locations. Zope figures out which one you want by first looking in the folder that you are calling the object on. If it can't find the object there it goes up one level and looks in the folder's containing folder. If the object can't be found there it goes up another level. This process continues until Zope finds the object or gets to the root folder. If Zope can't find the object in the root it gives up and raises an exception.

You'll see this kind of dynamic behavior in many different places in Zope. This technique is called *acquisition*. A folder is said to *acquire* a object by searching for the object in its containers.

The Special Folder Object *index_html*

As you've seen, folders can acquire all kinds of objects. There is one special object that Zope uses to display a folder. This object is named *index_html*.

The *index_html* object provides a default view of the folder. This is analogous to how an *index.html* file provides a default view for a directory in Apache and other web servers.

For example, if you create an *index_html* object in your *Invoices* folder and view the folder by clicking the View tab or by visiting the URL *http://localhost:8080/Invoices/*, Zope will call the *index_html* object on the *Invoices* folder.

A folder can also acquire an *index_html* object from its parent folders just as it can acquire any object. You can use this behavior to create a default view for a bunch of folders all in one place. If you want a different default view of a given folder, just create a custom *index_html* object in that folder. This way you can override the *index_html* object defined higher up.

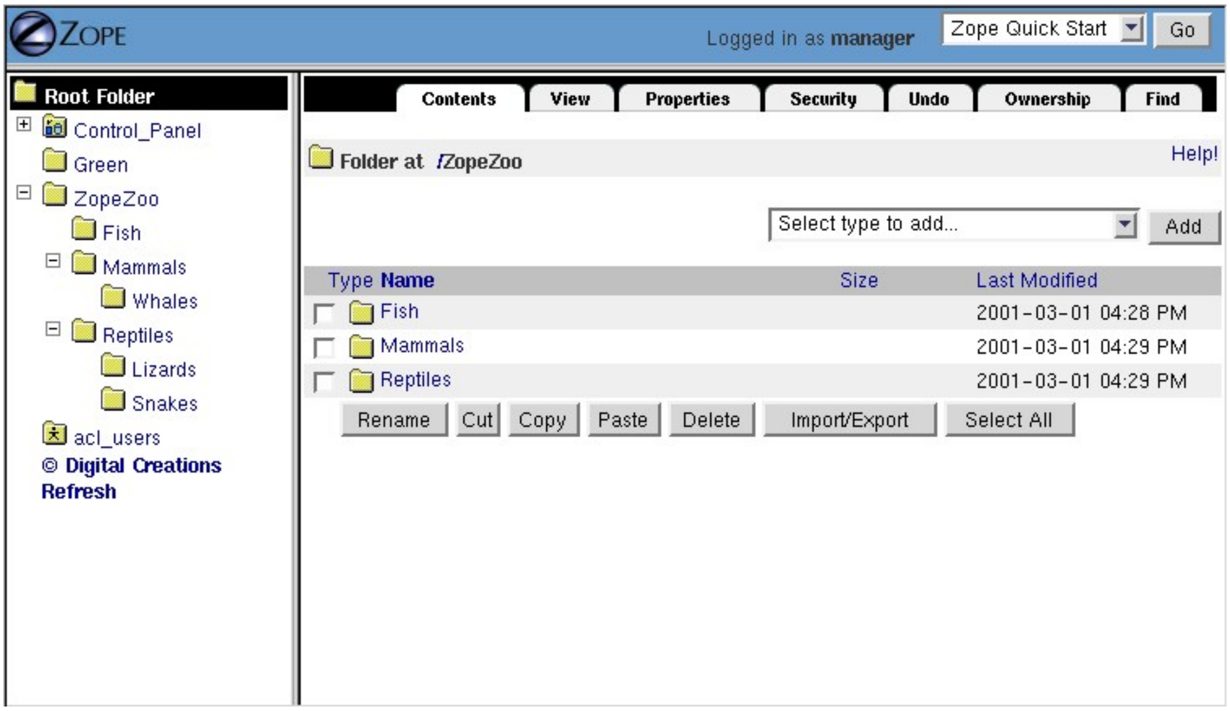
Building the Zope Zoo Website

In this section, you'll create a simple web site for the Zope Zoo. As the Zoo webmaster, it is your job to make the web site easy to use and manage. Here are some things you'll need:

- Zoo users must easily move around the site, just as if they were walking through a real Zoo.
- All of your shared web layout tools, like a Cascading Style Sheet (CSS), must be in one easy to manage location.
- You must provide a simple file library of various documents that describe the animals.
- You need a site map so that users can quickly get an idea of the layout of the entire Zoo.
- A Guest book must be created so that Zoo visitors can give you feedback and comments about your site.
- A what's new section must be added to the guest book so that you can see any recent comments that have been added.

Navigating the Zoo

In order for your navigation system to work, your site will need some basic structure through which to navigate. Create some folders in your Zope system that represent the structure of your site. Let's use a zoo structure with the following layout, as shown in Figure 5−1.



Folder at /ZopeZoo

Type Name	Size	Last Modified
Fish		2001-03-01 04:28 PM
Mammals		2001-03-01 04:29 PM
Reptiles		2001-03-01 04:29 PM

Figure 5−1 Zoo folder structure.

The main structure of the Zope Zoo contains three top level folders, *Reptiles*, *Mammals* and *Fish*. To navigate your site, users should first go to your home page and click on one of the top level folders to enter that particular part of the Zoo. They should also be able to use a very similar interface to keep going deeper into the site; i.e. the snakes section. Also, the user should be able to back out of a section and go up to the parent section.

You can accomplish this easily with Zope. In your *ZopeZoo* folder, create a DTML Method called *navigation*:

```
<ul>
<dtml-in expr="objectValues('Folder')">
  <li><a href="&dtml-absolute_url;"><dtml-var title_or_id></a></li><br>
</dtml-in>
</ul>
```

The method you just created shows a list of links to the various sub−sections of the zoo. It's important to notice that this method can work on any zoo folder since it makes no assumptions about the folder. Also since we placed this method in the *ZopeZoo* folder, all the zoo folders can acquire it.

Now, you need to incorporate this method into the site. Let's put a reference to it in the *standard_html_header* object so that the navigation system is available on every page of the site. Your *standard_html_header* could look like this:

```
<html>
<head><title><dtml-var title></title></head>
<body>
<dtml-var navigation>
```

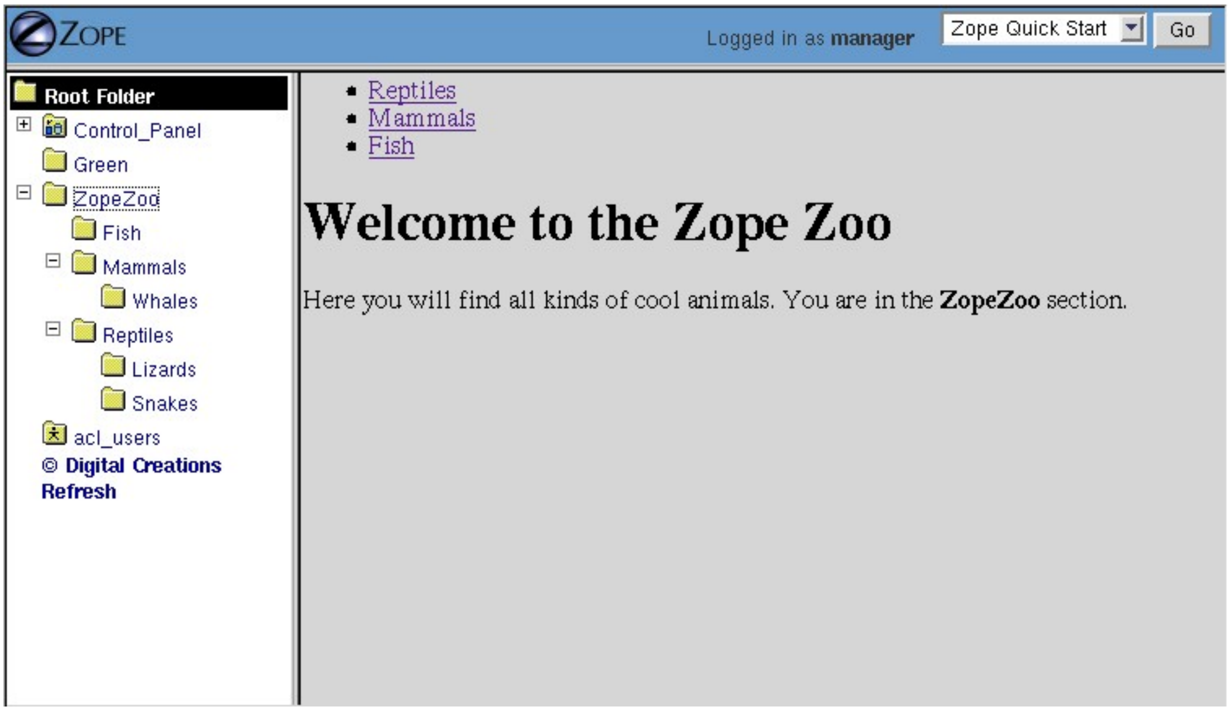
Next we need to add a front page to the Zoo site and then we can view the site and verify that the navigation works correctly.

Adding a Front Page to the Zoo

Now, you need a front page that serves as the welcome screen for Zoo visitors. Let's create a DTML Method in the *ZopeZoo* folder called *index_html* with the following content:

```
<dtml-var standard_html_header>

  <h1>Welcome to the Zope Zoo</h1>

  <p>Here you will find all kinds of cool animals.  You are in
  the <b><dtml-var getId></b> section.</p>

<dtml-var standard_html_footer>
```

Take a look at how your site appears by clicking on the *View* tab in the root folder, as shown in Figure 5−2.



- Reptiles
- Mammals
- Fish

Welcome to the Zope Zoo

Here you will find all kinds of cool animals. You are in the **ZopeZoo** section.

Figure 5−2 Zope Zoo front page.

Here you start to see how things come together. At the top of your main page you see a list of links to the various subsections. These links are created by the *navigation* method that is called by the *standard_html_header* method.

You can use the navigation links to travel through the various sections of the Zoo. Use this navigation interface to find the reptiles section.

Zope builds this page to display a folder by looking for the default folder view method ,*index_html*. It walks up the zoo site folder by folder until it finds the *index_html* method in the *ZopeZoo* folder. It then calls this method on the *Reptiles* folder. The *index_html* method calls the *standard_html_header* method which in turn calls the *navigation* method. Finally, the *index_html* method displays a welcome message and calls the *standard_html_footer*.

What if you want the reptile page to display something besides the welcome message? You can replace the *index_html* method in the reptile section with a more appropriate display method and still take advantage of the zoo header and footer including navigation.

In the *Reptile* folder create a DTML Method named *index_html*. Give it some content more appropriate to reptiles:

```
<dtml-var standard_html_header>

<h1>The Reptile House</h1>

<p>Welcome to the Reptile House.</p>

<p>We are open from 6pm to midnight Monday through Friday.</p>

<dtml-var standard_html_footer>
```

Now take a look at the reptile page by going to the *Reptile* folder and clicking the View tab.

Since the *index_html* method in the *Reptile* folder includes the standard headers and footers, the reptile page still includes your navigation system.

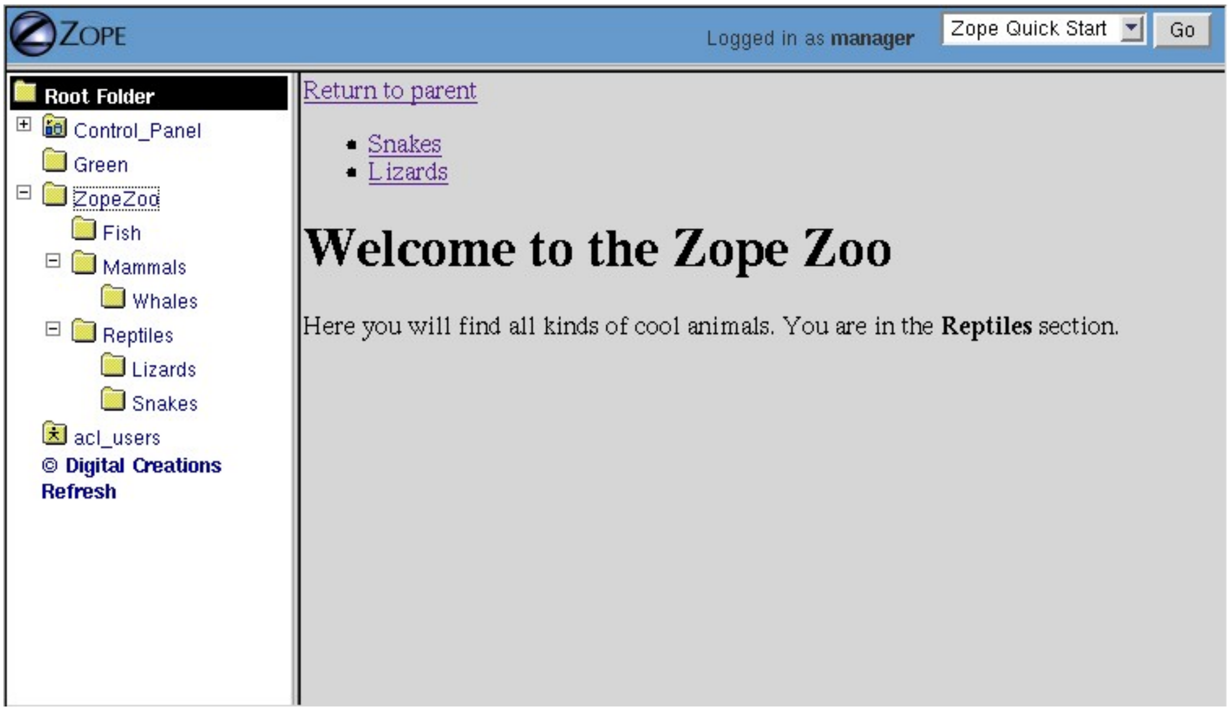
Click on the *Snakes* link on the reptile page to see what the Snakes section looks like. The snakes page looks like the *Reptiles* page because the *Snakes* folder acquires its *index_html* display method from the *Reptiles* folder.

Improving Navigation

The navigation system for the zoo works pretty well, but it has one big problem. Once you go deeper into the site you need to use your browser's *back* button to go back. There are no navigation links to allow you to navigate up the folder hierarchy. Let's add a navigation link to allow you to go up the hierarchy. Change the *navigation* method in the root folder:

```
<a href="..">Return to parent</a><br>

<ul>
<dtml-in expr="objectValues('Folder')">
  <li><a href="&dtml-absolute_url;"><dtml-var title_or_id></a><br></li>
</dtml-in>
</ul>
```

Now browse the Zoo site to see how this new link works, as shown in Figure Figure 5−3.



Return to parent

- Snakes
- Lizards

Welcome to the Zope Zoo

Here you will find all kinds of cool animals. You are in the **Reptiles** section.

Figure 5−3 Improved zoo navigation controls.

As you can see, the *Return to parent* link allows you to go back up from a section of the site to its parent. However some problems remain; when you are at the top level of the site you still get a *Return to parent* link which leads nowhere. Let's fix this by changing the *navigation* method to hide the parent link when you're in the *ZopeZoo* folder:

```
<dtml-if expr="_.len(PARENTS) > 2">
  <a href="..">Return to parent</a><br>
</dtml-if>

<ul>
<dtml-in expr="objectValues('Folder')">
  <li><a href="&dtml-absolute_url;"><dtml-var title_or_id></a><br></li>
</dtml-in>
</ul>
```

Now the method tests to see if the current object has any parents before it display a link to the parent. *PARENTS* is a list of the current object's parents, and *len* is a utility function which returns the length of a list. See Appendix A for more information on DTML utility functions. Now view the site. Notice that now there is no parent link when you're viewing the main zoo page.

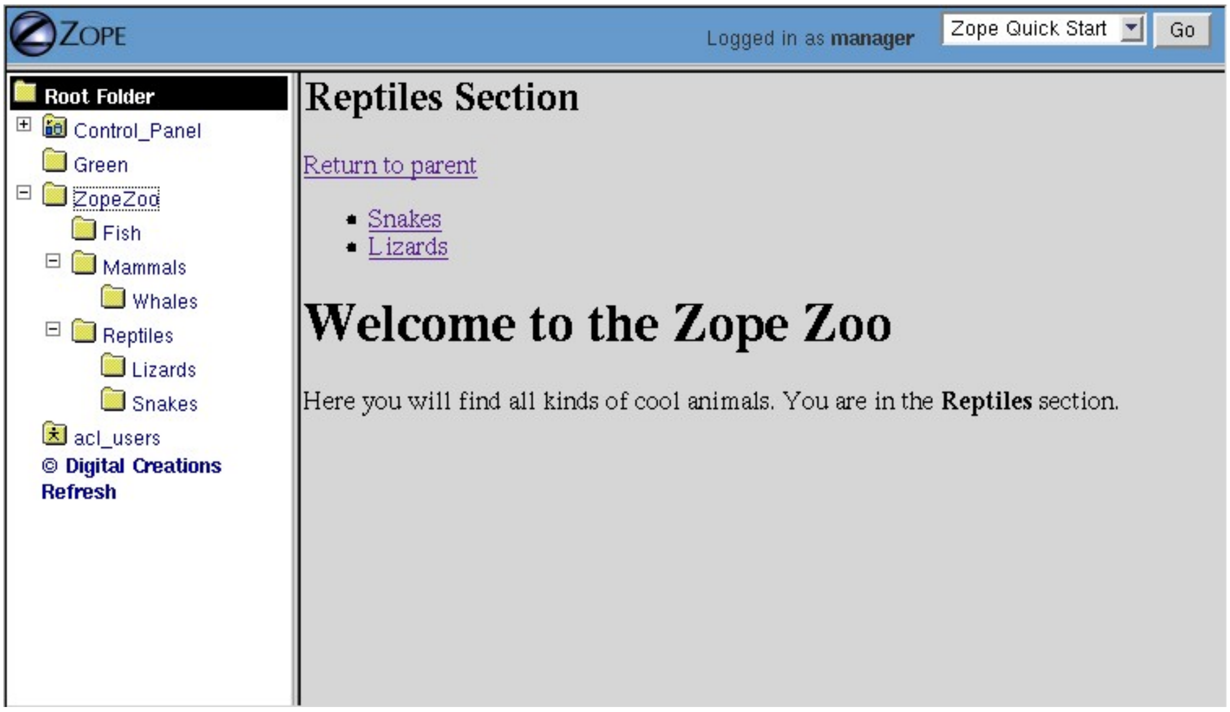
There are still some things that could be improved about the navigation system. For example, it's pretty hard to tell what section of the Zoo you're in. You've changed the reptile section, but the rest of the site all looks pretty much the same with the exception of having different navigation links. It would be nice to have each page tell you what part of the Zoo you're in.

Let's change the *navigation* method once again to display where you are:

```
<dtml-if expr="_.len(PARENTS) > 2">
  <h2><dtml-var title_or_id> Section</h2>
  <a href="..">Return to parent</a><br>
</dtml-if>
```

```
        <ul>
        <dtml-in expr="objectValues('Folder')">
          <li><a href="&dtml-absolute_url;"><dtml-var title_or_id></a><br></li>
        </dtml-in>
        </ul>
```

Now view the site again.



Reptiles Section

Return to parent

- Snakes
- Lizards

Welcome to the Zope Zoo

Here you will find all kinds of cool animals. You are in the **Reptiles** section.

Figure 5−4 Zoo page with section information.

As you can see in Figure 5−4, the navigation method now tells you what section you're in along with links to go to different sections of the zoo.

Factoring out Style Sheets

Zoo pages are built by collections of methods that operate on folders. For example, the header method calls the navigation method to display navigation links on all pages. In addition to factoring out shared behavior such as navigation controls, you can use different Zope objects to factor out shared content.

Suppose you'd like to use CSS (Cascading Style Sheets) to tailor the look and feel of the zoo site. One way to do this would be to include the CSS tags in the *standard_html_header* method. This way every page of the site would have the CSS information. This is a good way to reuse content, however, this is not a flexible solution since you may want a different look and feel in different parts of your site. Suppose you want the background of the snakes page to be green, while the rest of the site should have a white background. You'd have to override the *standard_html_header* in the *Snakes* folder and make it exactly the same as the normal header with the exception of the style information. This is an inflexible solution since you can't vary the CSS information without changing the entire header.

You can create a more flexible way to define CSS information by factoring it out into a separate object that the header will insert. Create a DTML Document in the *ZopeZoo* folder named *style_sheet*. Change the contents of the document to include some style information:

```
<style type="text/css">
h1{
  font-size: 24pt;
  font-family: sans-serif;
}
p{
  color: #220000;
}
body{
  background: #FFFFDD;
}
</style>
```

This is a CSS style sheet that defines how to display *h1*, *p* and *body* HTML tags. Now let's include this content into our web site by inserting it into the *standard_html_header* method:

```
<html>
<head>
<dtml-var style_sheet>
</head>
<body>
<dtml-var navigation>
```

Now, when you look at documents on your site, all of their paragraphs will be dark red, and the headers will be in a sans−serif font.

To change the style information in a part of the zoo site, just create a new *style_sheet* document and drop it into a folder. All the pages in that folder and its sub−folders will use the new style sheet.

Creating a File Library

File libraries are common on web sites since many sites distribute files of some sort. The old fashioned way to create a file library is to upload your files, then create a web page that contains links to those files. With Zope you can dynamically create links to files. When you upload, change or delete files, the file library's links can change automatically.

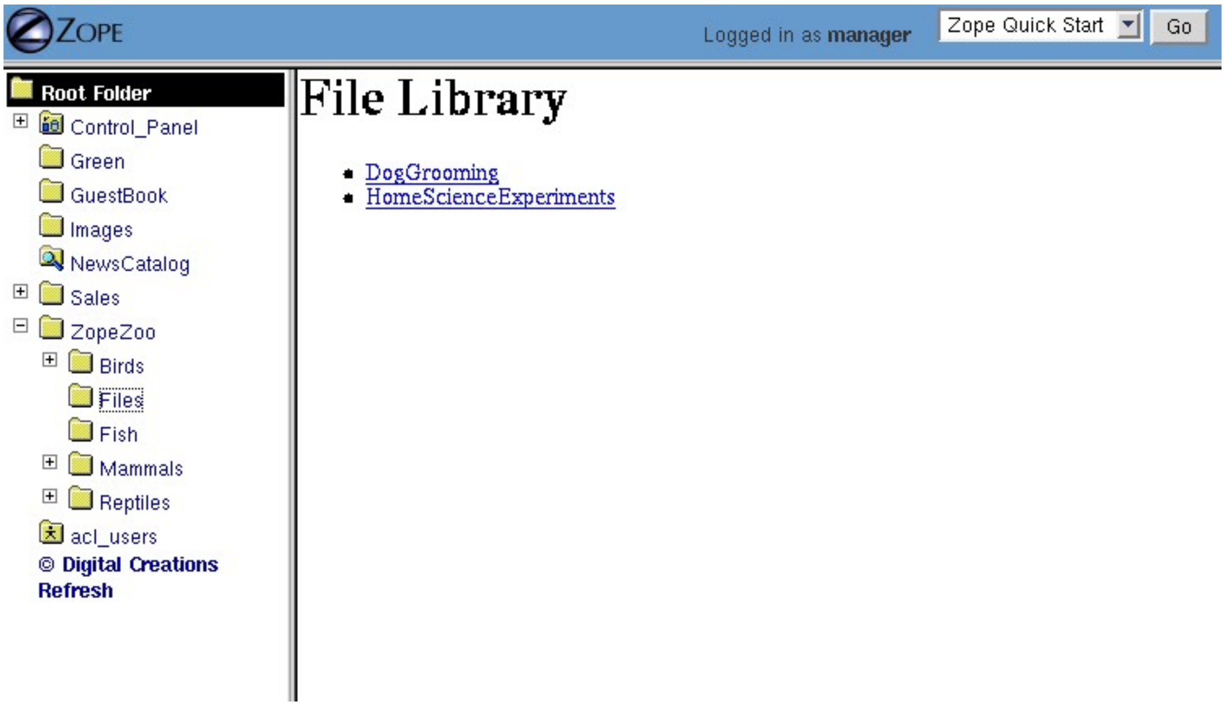
Create a folder in the *ZopeZoo* folder called *Files*. This folder contains all of the file you want to distribute to your web visitors.

In the *Files* folder create some empty file objects with names like *DogGrooming* or *HomeScienceExperiments*, just to give you some sample data to work with. Add some descriptive titles to these files.

DTML can help you save time maintaining this library. Create an *index_html* DTML Method in the *Files* folder to list all the files in the library:

```
<dtml-var standard_html_header>

<h1>File Library</h1>

<ul>
<dtml-in expr="objectValues('File')">
  <li><a href="&dtml-absolute_url;"><dtml-var title_or_id></a></li>
</dtml-in>
</ul>

<dtml-var standard_html_footer>
```

Now view the *Files* folder. You should see a list of links to the files in the *Files* folder as shown in Figure 5−5.



File Library

- DogGrooming
- HomeScienceExperiments

Figure 5−5 File library contents page.

If you add another file, Zope will dynamically adjust the file library page. You may also want to try changing the titles of the files, uploading new files, or deleting some of the files.

The file library as it stands is functional but Spartan. The library doesn't let you know when a file was created, and it doesn't let you sort the files in any way. Let's make the library a little fancier.

Most Zope objects have a *bobobase_modification_time* method that returns the time the object was last modified. We can use this method in the file library's *index_html* method:

```
<dtml-var standard_html_header>

<h1>File Library</h1>

<table>
  <tr>
    <th>File</th>
    <th>Last Modified</th>
   </tr>

<dtml-in expr="objectValues('File')">
  <tr>
     <td><a href="&dtml-absolute_url;"><dtml-var title_or_id></a></td>
     <td><dtml-var bobobase_modification_time fmt="aCommon"></td>
  </tr>
</dtml-in>

</table>

<dtml-var standard_html_footer>
```

The new file library method uses an HTML table to display the files and their modification times.

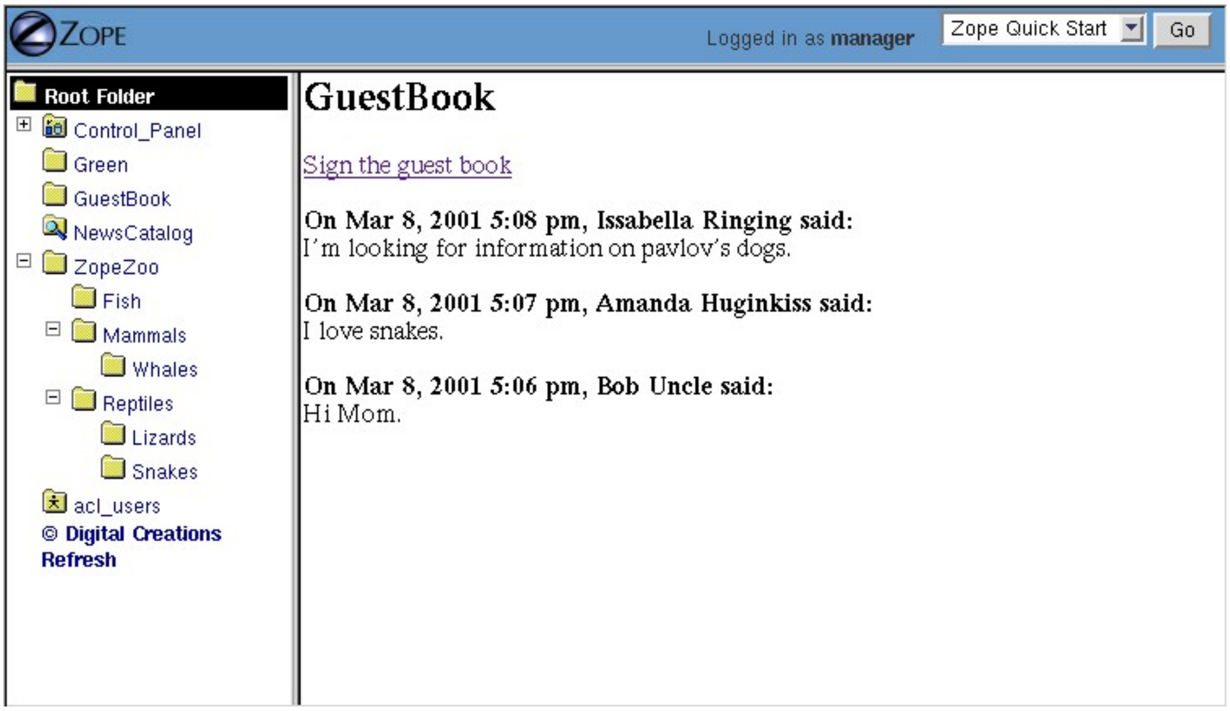
Finally let's add the ability to sort this list by file name or by modification date. Change the *index_html* method again:

```
<dtml-var standard_html_header>

<h1>File Library</h1>

<table>
  <tr>
    <th><a href="&dtml-URL0;?sort=name">File</a></th>
    <th><a href="&dtml-URL0;?sort=date">Last Modified</a></th>
   </tr>

<dtml-if expr="_.has_key('sort') and sort=='date'">
  <dtml-in expr="objectValues('File')"
           sort="bobobase_modification_time" reverse>
    <tr>
       <td><a href="&dtml-absolute_url;"><dtml-var title_or_id></a></td>
       <td><dtml-var bobobase_modification_time fmt="aCommon"><td>
    </tr>
  </dtml-in>
<dtml-else>
  <dtml-in expr="objectValues('File')" sort="id">
    <tr>
       <td><a href="&dtml-absolute_url;"><dtml-var title_or_id></a></td>
       <td><dtml-var bobobase_modification_time fmt="aCommon"><td>
    </tr>
  </dtml-in>
</dtml-if>

</table>

<dtml-var standard_html_footer>
```

Now view the file library and click on the *File* and *Last Modified* links to sort the files. This method works with two sorting loops. One uses the *in* tag to sort on an object's *id*. The other does a reverse sort on an object's *bobobase_modification_time* method. The *index_html* method decides which loop to use by looking for the *sort* variable. If there is a *sort* variable and if it has a value of *date* then the files are sorted by modification time. Otherwise the files are sorted by id.

Building a Guest Book

A guest book is a common and useful web application that allows visitors to your site to leave messages. Figure Figure 5−6 shows what the guest book you're going to write looks like.



GuestBook

Sign the guest book

On Mar 8, 2001 5:08 pm, Issabella Ringing said:
I'm looking for information on pavlov's dogs.

On Mar 8, 2001 5:07 pm, Amanda Huginkiss said:
I love snakes.

On Mar 8, 2001 5:06 pm, Bob Uncle said:
Hi Mom.

Figure 5−6 Zoo guest book.

Start by creating a folder called *GuestBook* in the root folder. Give this folder the title `The Zope Zoo Guest Book`. The *GuestBook* folder will hold the guest book entries and methods to view and add entries. The folder will hold everything the guest book needs. After the guest book is done you will be able to copy and paste it elsewhere in your site to create new guest books.

You can use Zope to create a guest book several ways, but for this example, you'll use one of the simplest. The *GuestBook* folder will hold a bunch of DTML Documents, one document for each guest book entry. When a new entry is added to the guest book, a new document is created in the *GuestBook* folder. To delete an unwanted entry, just go into the *GuestBook* folder and delete the unwanted document using the management interface.

Let's create a method that displays all of the entries. Call this method *index_html* so that it is the default view of the *GuestBook* folder:

```
<dtml-var standard_html_header>

<h2><dtml-var title_or_id></h2>

<!-- Provide a link to add a new entry, this link goes to the
addEntryForm method -->

<p>
  <a href="addEntryForm">Sign the guest book</a>
</p>

<!-- Iterate over each DTML Document in the folder starting with
the newest documents first. -->

<dtml-in expr="objectValues('DTML Document')"
         sort="bobobase_modification_time" reverse>

<!-- Display the date, author and contents of each document -->
```

```
  <p>
  <b>On <dtml-var bobobase_modification_time fmt="aCommon">,
     <dtml-var guest_name html_quote null="Anonymous"> said:</b><br>

  <dtml-var sequence-item html_quote newline_to_br>

  <!-- Make sure we use html_quote so the users can't sneak any
  HTML onto our page -->

</p>

</dtml-in>

<dtml-var standard_html_footer>
```

This method loops over all the documents in the folder and displays each one. Notice that this method assumes that each document will have a *guest_name* property. If that property doesn't exist or is empty, then Zope will use *Anonymous* as the guest name. When you create a entry document you'll have to make sure to set this property.

Next, let's create a form that your site visitors will use to add new guest book entries. In the *index_html* method above we already created a link to this form. In your *GuestBook* folder create a new DTML Method named *addEntryForm*:

```
<dtml-var standard_html_header>

<p>Type in your name and your comments and we'll add it to the
guest book.</p>

<form action="addEntryAction" method="POST">
<p> Your name:
  <input type="text" name="guest_name" value="Anonymous">
</p>
<p> Your comments: <br>
  <textarea name="comments" rows="10" cols="60"></textarea>
</p>

<p>
  <input type="submit" value="Send Comments">
</p>
</form>

<dtml-var standard_html_footer>
```

Now when you click on the *Sign Guest Book* link on the guest book page you'll see a form allowing you to type in your comments. This form collects the user's name and comments and submits this information to a method named *addEntryAction*.

Now create an *addEntryAction* DTML Method in the *GuestBook* folder to handle the form. This form will create a new entry document and return a confirmation message:

```
<dtml-var standard_html_header>

<dtml-call expr="addEntry(guest_name, comments)">

<h1>Thanks for signing our guest book!</h1>

<p><a href="<dtml-var URL1>">Return</a>
to the guest book.</p>
```

```
        <dtml-var standard_html_footer>
```

This method creates a new entry by calling the *addEntry* method and returns a message letting the user know that their entry has been added.

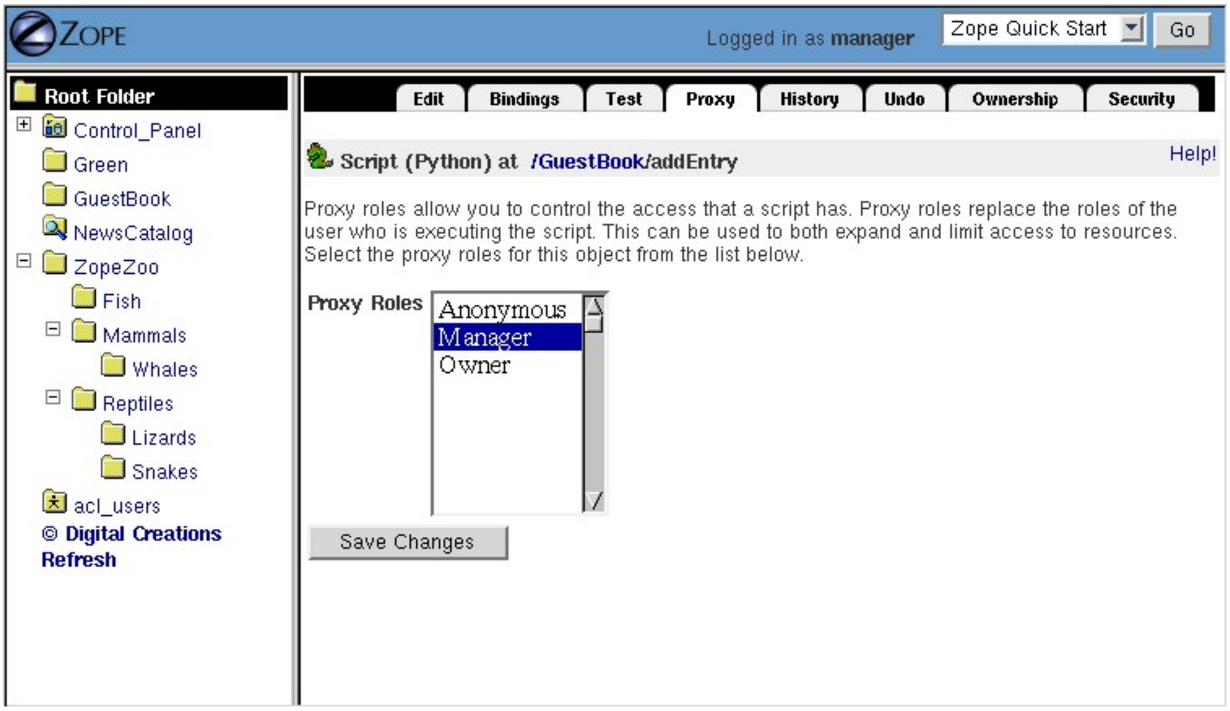
The last remaining piece of the puzzle is to write the script that will create a document and sets its contents and properties. We'll do this in Python since it is much clearer than doing it in DTML. Create a Python−based Script in the *GuestBook* folder called *addEntry* with parameters *guest_name* and *comments*:

```
        ## Script (Python) "addEntry"
        ##parameters=guest_name, comments
        ##
        """
        Create a guest book entry.
        """
        # create a unique document id
        id='entry_%d' % len(context.objectIds())

        # create the document
        context.manage_addProduct['OFSP'].manage_addDTMLDocument(id,
                                                 title="", file=comments)

        # add a guest_name string property
        doc=getattr(context, id)
        doc.manage_addProperty('guest_name', guest_name, 'string')
```

This script uses Zope API calls to create a DTML Document and to create a property on that document. This script performs the same sort of actions in a script that you could do manually; it creates a document, edits it and sets a property.

The guest book is now almost finished. To use the simple guest book, just visit *http://localhost:8080/GuestBook/*.

One final thing is needed to make the guest book complete. More than likely your security policy will not allow anonymous site visitors to create documents. However the guest book application should be able to be used by anonymous visitors. In Chapter 7, User and Security, we'll explore this scenario more fully. The solution is to grant special permission to the *addEntry* method to allow it to do its work of creating a document. You can do this by setting the *Proxy role* of the method to *Manager*. This means that when the method runs it will work as though it was run by a manager regardless of who is actually running the method. To change the proxy roles go to the *Proxy* view of the *addEntry* method, as shown in Figure 5−7.



Script (Python) at /GuestBook/addEntry

Proxy roles allow you to control the access that a script has. Proxy roles replace the roles of the user who is executing the script. This can be used to both expand and limit access to resources. Select the proxy roles for this object from the list below.

Proxy Roles: Anonymous, Manager (selected), Owner

Figure 5−7 Setting proxy roles for the addEntry method.

Now select *Manager* from the list of proxy roles and click *Change*.

Congratulations, you've just completed a functional web application. The guest book is complete and can be copied to different sites if you want.

Extending the Guest Book to Generate XML

All Zope objects can create XML. It's fairly easy to create XML with DTML. XML is just a way of describing information. The power of XML is that it lets you easily exchange information across the network. Here's a simple way that you could represent your guest book in XML:

```
<guestbook>
  <entry>
    <comments>My comments</comments>
  </entry>
  <entry>
    <comments>I like your web page</comments>
  </entry>
  <entry>
    <comments>Please no blink tags</comments>
  </entry>
</guestbook>
```

This XML document may not be that complex but it's easy to generate. Create a DTML Method named "entries.xml" in your guest book folder with the following contents:

```
<guestbook>
  <dtml-in expr="objectValues('DTML Document')">
  <entry>
    <comments><dtml-var document_src html_quote></comments>
  </entry>
  </dtml-in>
</guestbook>
```

As you can see, DTML is equally adept at creating XML as it is at creating HTML. Simply embed DTML tags among XML tags and you're set. The only tricky thing that you may wish to do is to set the content−type of the response to *text/xml*, which can be done with this DTML code:

```
        <dtml−call expr="RESPONSE.setHeader('content−type', 'text/xml')">
```

The whole point of generating XML is producing data in a format that can be understood by other systems. Therefore you will probably want to create XML in an existing format understood by the systems you want to communicate with. In the case of the guest book a reasonable format may be the RSS (Rich Site Summary) XML format. RSS is a format developed by Netscape for its *my.netscape.com* site, which has since gained popularity among other web logs and news sites. The Zope.org web site uses DTML to build a dynamic RSS document.

Congratulations! You've XML−enabled your guest book in just a couple minutes. Pat yourself on the back. If you want extra credit, research RSS enough to figure out how to change *entries.xml* to generate RSS.

The Next Step

This chapter shows how simple web applications can be made. Zope has many more features in addition to these, but these simple examples should get you started on create well managed, complex web sites.

In the next chapter, we'll see how the Zope security system lets Zope work with many different users at the same time and allows them to collaborate together on the same projects.

Chapter 7: Users and Security

All web applications need to manage security. Managing security means controlling who can access your application, and determining what they can do. Security is not an afterthought that can be added to protect a working system. Instead security should be an important design element that you consider as you build your Zope applications.

In this chapter you'll find out how to create and manage user accounts, and how to control how users can use your application by creating security policies.

Introducing Security

Security controls what the users of your site can do and how you and others can maintain your site. If you carefully consider security you can provide powerful features to your users and allow large groups of people to safely work together to maintain your site. If you do not consider security, it will be difficult to give your users control safely and managing your site will become a messy burden. Your site will suffer not only from people doing harmful things that they shouldn't be able to do, but it will be hard for you to provide value to your users and control to those who manage your site.

Zope weaves security into almost every aspect of web application building. Zope uses the same security system to control Zope management as you use to create users for your application. Zope makes no distinction between using and managing an application. This may seem confusing, but in fact it allows you to leverage Zope's security framework for your application's needs.

Logging in and Logging Out of Zope

As we saw in Chapter 2, "Using Zope" you log into Zope by going to a management URL in your web browser and entering your username and password. We also pointed out in Chapter 2, "Using Zope" that because of how most web browsers work, you must quit your browser to log out of Zope.

If you attempt to access a protected resource for which you don't have access privileges, Zope will prompt you to log in. This can happen even if you're already logged in. In general, there is no need to log in to Zope if you only wish to use public resources.

Authentication and Authorization

Security in the broadest sense controls two functions, *authentication* and *authorization*. Authentication means finding out who you are, and authorization means determining what you can do. Zope provides separate facilities to manage the processes of identifying users and granting access to controlled actions.

When you access a protected resource (for example, by viewing a private web page) Zope will ask you to log in and will look for your user account. This is the authentication process. Note that Zope will only authenticate you if you ask for a protected resource; if you only access public resources Zope will continue to assume you are anonymous.

Once you've been authenticated, Zope determines whether or not you have access to the protected resource. This process involves two intermediary layers between you and the protected resource, *roles* and *permissions*. Users have roles which describe what they can do such as "Author", "Manager", and "Editor". Zope objects have permissions which describe what can be done with them such as "View", "Delete objects", and "Manage properties".

Security policies map roles to permissions; in other words they say who can do what. For example, a security policy may associate the "Manager" role with the "Delete objects" permission. This allows managers to delete objects. In this way Zope authorizes users to perform protected actions.

In the following sections we'll look more closely at authentication and authorization and how to effectively set security policies. First you'll learn about authentication using users and User Folders, then you'll find out about controlling authorization with security policies.

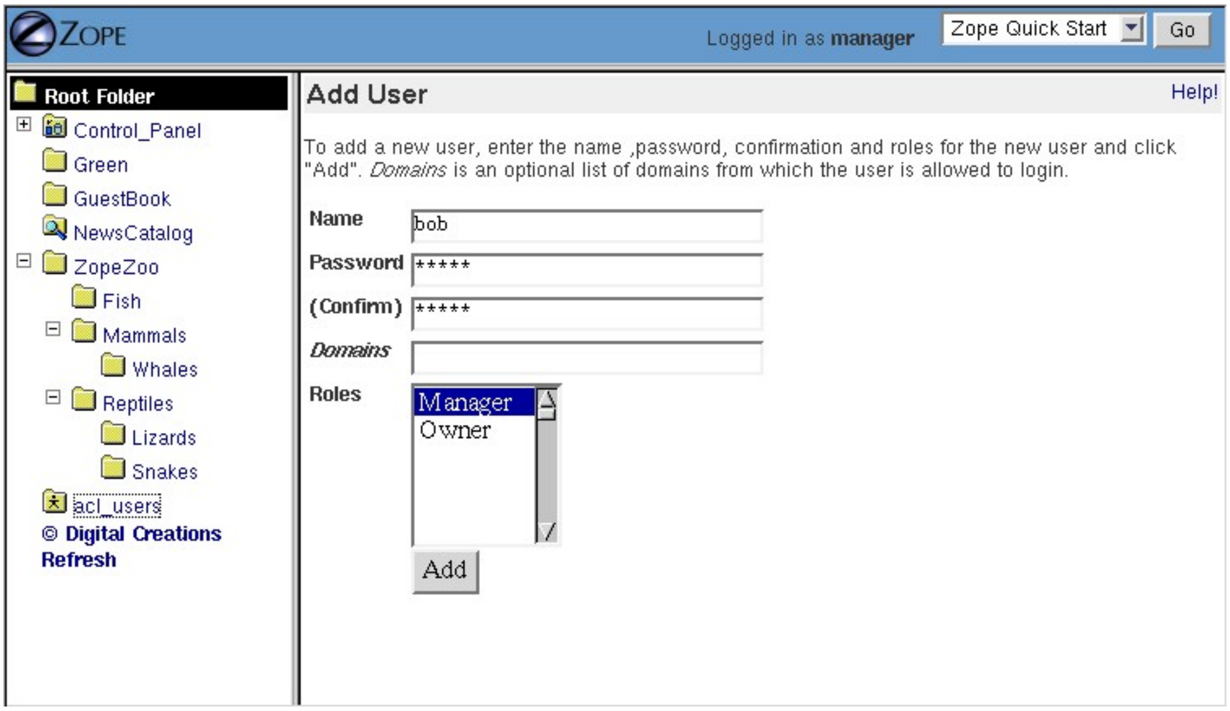
Authentication and Managing Users

A Zope *User* defines a user account. A Zope user has a name, a password, and optionally additional data about someone who uses Zope. To log into Zope, you must have a user account. Let's examine how to create and manage user accounts.

Creating Users in User Folders

To create user accounts in Zope you add users to user folders. In Chapter 2, "Using Zope" you should already have added a Manager user to your top level user folder.

Let's create a new user so that your coworker can help you manage your Zope site. Go to the root Zope folder. Click on the user folder named *acl_users*. The user folder contains user objects that define Zope user accounts. Click the *Add* button to create a new user.



Add User

To add a new user, enter the name ,password, confirmation and roles for the new user and click "Add". *Domains* is an optional list of domains from which the user is allowed to login.

Name	bob
Password	*****
(Confirm)	*****
Domains	

Figure 6−1 Adding a user to a user folder.

The form in Figure 6−1 lets you define the user. Type a username for your coworker in the *Name* field, for example, "michel". The username can contain letters, spaces, and numbers. The username is case sensitive.

Choose a password for your coworker and enter it in the *Password* and *(Confirm)* fields. We'll set things up so that your coworker can change their password later when they log in. You might want to use a password like "change me" to help remind them to change their password.

The *Domains* field lets you restrict Internet domains from which the user can log in. This allows you to add another safety control to your account. For example if you always want your coworker to log in from work you could enter your work's domain, for example "myjob.com", in the Domains field. You can specify multiple domains separated by spaces to allow the user to log in from multiple domains. For example if you decide that your coworker should be able to manage Zope from their home account too, you could set the domains to "myjob.com myhome.net". You can also use IP numbers with asterisks to indicate wild card instead of domain names to specify domains. For example, "209.67.167.*" will match all IP addresses that start with "209.67.167".

The *Roles* select list indicates which roles the user should have. In general users who are performing management tasks should be given the *Manager* role. In the case of your coworker, select the Manager role. The *Owner* role is not appropriate in most cases because a user is normally an owner of a specific object, not an owner in general. We'll look at ownership more later in the chapter. We'll also see later how you can define your own roles such as *Editor* and *Reviewer*.

To create the new user click the *Add* button. You should see a new user object in the user folder.

Editing Users

You can edit existing users by clicking on them. This displays a form very similar to the form you used to create a user. In fact you can control all the same settings that we just discussed from this form. After your coworker logs in with the account you created for them, they should go to this management screen and change their password here.

Like all Zope management functions, editing users is protected by the security policy. A user can only change their password if they have the *Manage Users* permission, which managers have by default.

So by default it's possible for a manager defined in a given user folder to change other managers' accounts if they both are defined in the same user folder. This may or may not be what you want. Later we'll look at ways to avoid this potential problem. Rest assured that it is not, however, possible for someone to find out your password from the management interface. Another manager may have access to *change* your password, but not find out what your current password is without changing it.

In general, user folders work like normal Zope folders; you can create, edit and delete contained objects. However, user folders are not as capable as normal folders. You cannot cut and paste users in a user folder, and you can't create anything besides a user in a user folder.

To delete an existing user from a user folder, select the user and click the *Delete* button. Remember, like all Zope actions, this can be undone if you make a mistake.

Defining a User's Location

Zope can contain multiple user folders at different locations in the object hierarchy. A Zope user cannot access resources above the user folder they are defined in. Where your user account is defined determines what Zope resources you can access.

If your account is defined in a user folder in the root folder, you have access to the root folder. This is probably where the account you are using right now is defined. You can however, define users in any Zope folder.

Consider the case of a user folder at */BeautySchool/Hair/acl_users*. Suppose the user *Ralph Scissorhands* is defined in this user folder. Ralph cannot log into to Zope above the folder at */BeautySchool/Hair*. Effectively

Ralph's view of the Zope site is limited to things in the *BeautySchool/Hair* folder and below. Regardless of the roles assigned to Ralph, he cannot access protected resources above his location.

Using this technique it's easy to build simple security policies. One of the most common Zope management patterns is to place related objects in a folder together and then create a user folder in that folder to define people who are responsible for those objects.

For example suppose people in your organization wear uniforms. You are creating an intranet that provides information about your organization, including information about uniforms. You might create a *uniforms* folder somewhere in the intranet Zope site. In that folder you could put objects such as pictures of uniforms and descriptions for how to wear and clean them. Then you could create a user folder in the uniforms folder and create an account for the head tailor. When a new style of uniform comes out the tailor doesn't have to ask the web master to update the site, he or she can update their own section of the site without bothering anyone else. Additionally, the head tailor cannot log into any folder above the *uniforms* folder, which means the head tailor cannot manage any objects other than those in the *uniforms* folder.

This security pattern is called *delegation*, and is very common in Zope applications. By delegating different areas of your Zope site to different users, you can take the burden of site administration off of a small group of managers and spread that burden around to different specific groups of users. Later in the chapter we'll look at other security patterns.

Working with Alternative User Folders

It may be that you don't want to manage your user account through the web. This may be because you already have a user database, or perhaps you want to use other tools to maintain your account information. Zope allows you to use all sorts of authentication techniques with alternate user folders. You can find many alternate user folders on the Zope web site at *http://www.zope.org/Products/user_management*. At the time of this writing there are 19 contributed alternate user folders. Here is a sampling of some of the more popular alternative user folders available.

LoginManager
: This is a flexible and powerful user folder that allows you to plug in your own authorization methods. For example, you can use LoginManager to authenticate from a database.

etcUserFolder
: This user folder authenticates using standard Unix */etc/password* style files.

LDAPAdapter
: This user folder allows you to authenticate from an LDAP server.

NTUserFolder
: This user folder authenticates from NT user accounts. It only works if you are running Zope under Windows NT or Windows 2000.

Some user folders provide alternate log in and log out controls such as log in web forms, rather than browser HTTP authorization controls. Despite this variety, all user folders use the same general log in procedure of prompting you for credentials when you access a protected resource.

While most users are managed with user folders of one kind or another, Zope has a few special user accounts that are not managed with user folder.

Special User Accounts

Zope provides three special user accounts which are not defined with user folders, the *anonymous user*, the *emergency user*, and the *initial manager*. The anonymous user is used frequently, while the emergency user and initial manager accounts are rarely used but are important to know about.

Zope Anonymous User

Zope has a built−in user account for guests, the anonymous user. If you don't have a user account on Zope, you'll be considered to be the anonymous user.

The anonymous user has security controls like any other, it has the role *Anonymous*. By default the *Anonymous* role can only access public resources, and can't change any Zope objects. You can tailor this policy, but most of the time you'll find the default anonymous security settings adequate.

As we mentioned earlier in the chapter, you must try to access a protected resource in order for Zope to authenticate you. The upshot is that even if you have a user account, Zope will consider you anonymous until you have logged in.

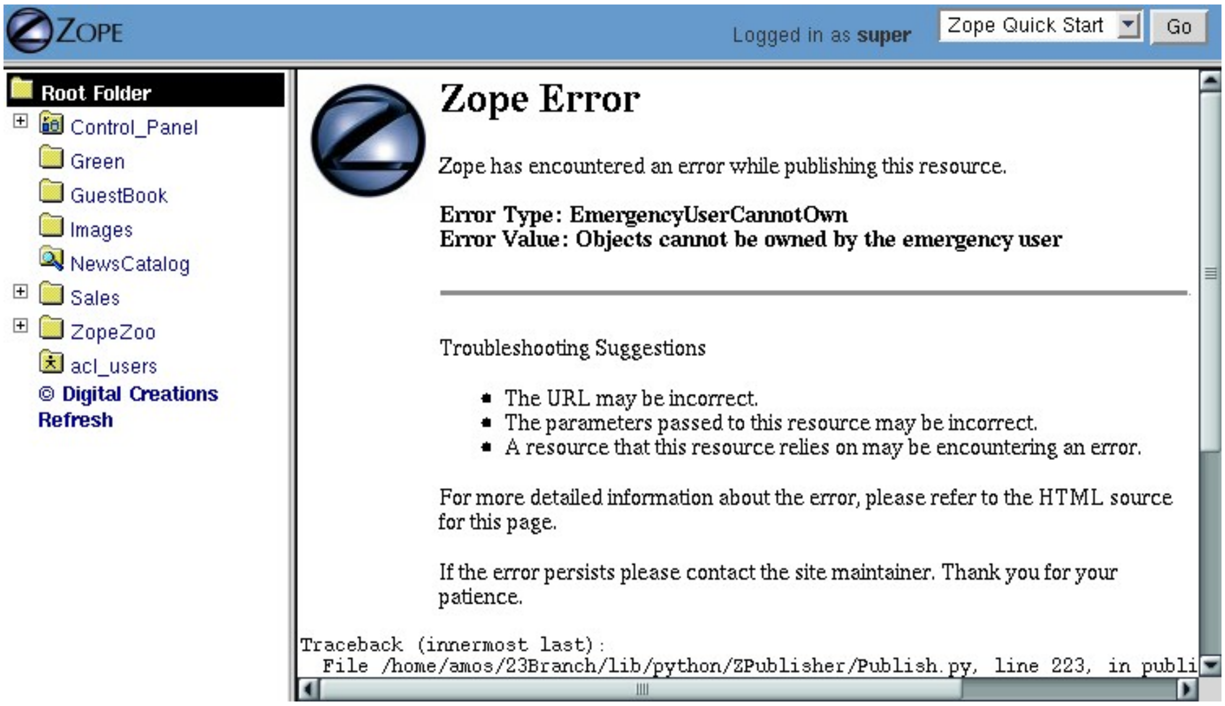
Zope Emergency User

Zope has a special user account for emergency use known as the *emergency user*. We discussed the emergency user briefly in Chapter 2, "Using Zope". The emergency user is not restricted by normal security settings. However, the emergency user cannot create any new objects with the exception of new user objects.

The emergency user is only really useful for two things: fixing messed up permissions, and creating manager accounts. As we saw in Chapter 2, "Using Zope" you can log in as the emergency user to create a manager account when none exist. After you create a manager account you should log out as the emergency user and log back in as the manager.

Another reason to use the emergency user account is if you lock yourself out of Zope by removing permissions you need to manage Zope. In this case log in as the emergency user and make sure that your manager account has the 'View management screens' and `Change permissions` permissions. Then log out and log back with your manager account and you should have enough access to fix anything else that is broken.

A common problem with the emergency user is trying to create a new object.



Zope Error

Zope has encountered an error while publishing this resource.

Error Type: EmergencyUserCannotOwn
Error Value: Objects cannot be owned by the emergency user

Troubleshooting Suggestions

- The URL may be incorrect.
- The parameters passed to this resource may be incorrect.
- A resource that this resource relies on may be encountering an error.

For more detailed information about the error, please refer to the HTML source for this page.

If the error persists please contact the site maintainer. Thank you for your patience.

```
Traceback (innermost last):
  File /home/amos/23Branch/lib/python/ZPublisher/Publish.py, line 223, in publi
```

Figure 6−2 Error caused by trying to create a new object when logged in as the Emergency User.

The error in Figure 6−2 lets you know that the emergency user cannot create new objects. The reason for this is a bit complex but will become clearer later in the chapter when we cover ownership. The short version of the story is that it would be unsafe for the emergency user to create objects since they would not be subject to the same security constraints as other objects.

Creating an Emergency User

Unlike normal user accounts that are defined through the web. The Emergency User account is defined in the filesystem. You can change the Emergency User account by editing the *access* file in the Zope directory. Zope comes with a command line utility, *zpasswd.py* to manage the Emergency User account. Run *zpasswd.py* by passing it the access file path:

```
$ python zpasswd.py access

Username: superuser
Password:
Verify password:

Please choose a format from:

SHA − SHA−1 hashed password
CRYPT − UNIX−style crypt password
CLEARTEXT − no protection.

Encoding: SHA
Domain restrictions:
```

The *zpasswd.py* script steps you through the process of creating a Emergency User account. Note that when you type in your password it is not echoed to the screen. You can also run `zpasswd.py` with no arguments to get a list of command line options.

Zope Initial Manager

The Initial manager account is created by the Zope installer so you can log into Zope the first time. When you first install Zope you should see a message like this:

```
creating default inituser file
Note:
        The initial user name and password are 'admin'
        and 'IVX3kAwU'.

        You can change the name and password through the web
        interface or using the 'zpasswd.py' script.
```

This lets you know the initial manager's name and password. You can use this information to log in to Zope for the first time as a manager. For there you can create additional user accounts.

Initial users are defined in a similar way to the emergency user; they are defined in a file on the filesystem call *inituser*. The *zpasswd.py* program can be used to edit this file the same way it is used to edit the emergency user *access* file:

```
$ python zpasswd.py inituser

Username: bob
Password:
Verify password:

Please choose a format from:

SHA − SHA−1 hashed password
CRYPT − UNIX−style crypt password
CLEARTEXT − no protection.

Encoding: SHA
Domain restrictions:
```

This will create a new initial user called "bob" and set its password (the password is not echoed back to you when you type it in). When Zope starts, it checks this file for users and makes sure they can log into Zope. Normally, initial users are created by the Zope installer for you, and you shouldn't have to worry about changing them. If you want to create additional users, you'll do it through the Zope web management interface.

So far we've covered how users and user folders control authentication. Next we'll look at how to control authorization with security policies.

Authorization and Managing Security

Zope security policies control authorization; they define who can do what. Security policies describe which roles have which permissions. Roles label classes of users, and permissions protect objects. Thus, security policies define which classes of users (roles) can take what kinds of actions (permissions) in a given part of the site.

Rather than stating which specific user can take which specific action on which specific object, Zope allows you to define which kinds of users can take which kinds of action in which areas of the site. This sort of generalizing makes your security policies simple and more powerful. Of course, you can make exceptions to your policy for specific users, actions, and objects.

In the following sections we'll examine roles, permissions and security policies more closely with an eye to building simple and effective security policies.

Working with Roles

Zope users have *roles* that define what kinds of actions they can take. Roles define classes of users such as *Manager*, *Anonymous*, and *Authenticated*.

Roles are similar to UNIX groups in that they abstract groups of users. And like UNIX groups, Zope users can have more than one role.

Roles make it easier for you to manage security. Instead of defining what every single user can do, you can simply set a couple different security policies for different user roles.

Zope comes with four built−in roles:

Manager
: This role is used for users who perform standard Zope management functions such as creating and edit Zope folders and documents.

Anonymous
: The Zope Anonymous User has this role. This role should be authorized to view public resources. In general this role should not be allowed to change Zope objects.

Owner
: This role is assigned automatically to users in the context of objects they create. We'll cover ownership later in this chapter.

Authenticated
: This role is assigned automatically to users who have provided valid authentication credentials. This role means that Zope "knows" who a particular user is.

For basic Zope sites you can get by with Manager and Anonymous. For more complex sites you may want to create your own roles to classify your users into different groups.

Defining Roles

To create a new role go to the *Security* tab and scroll down to the bottom of the screen. Type the name of the new role in the *User defined role* field, and click *Add Role*. Role names should be short one or two word descriptions of a type of user such as "Author", "Site Architect", or "Designer". You should pick role names that are relevant to your application.

You can verify that your role was created, noticing that there is now a role column for your new role at the top of the screen. You can also delete a role by selecting the role from the select list at the bottom of the security screen and clicking the *Delete Role* button. You can only delete your own custom roles, you cannot delete any of the "stock" roles that come with Zope.

You should notice that roles can be used at the level where they are defined and below in the object hierarchy. So if you want to create a role that is appropriate for your entire site create it in the root folder.

In general roles should be applicable for large sections of your site. If you find yourself creating roles to limit access to parts of your site, chances are there are better ways to accomplish the same thing. For example you could simply change the security settings for existing roles on the folder you want to protect, or you could define users deeper in the object hierarchy to limit their access. Later in the chapter we'll look at more examples of how to define security policies.

Understanding Local Roles

Local roles are an advanced feature of Zope security. Users can be given extra roles when working with a certain object. If an object has local roles associated with a user then that user gets those additional roles while working with that object.

For example, if a user owns an object they are usually given the additional local role of *Owner* while working with that object. A user might not have the ability to edit DTML Methods in general, but for DTML Methods they own, the user could have access to edit the DTML Method through the *Owner* local role.

Local roles are a fairly advanced security control and are not needed very often. Zope's automatic control of the *Owner* local role is likely the only place you'll encounter local roles.

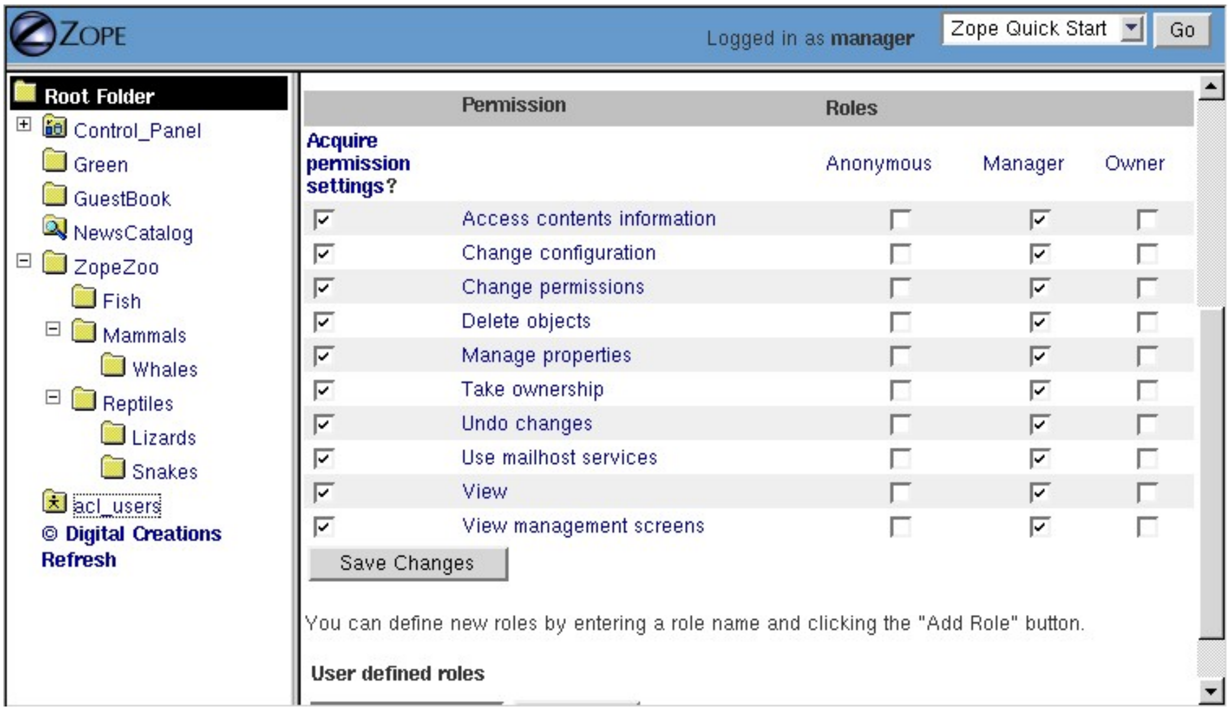
The main reason you might want to manually control local roles is to give a specific user special access to an object. In general you should avoid setting security for specific users if possible. It is easier to manage security settings that control groups of users instead of individuals.

Understanding Permissions

Permissions define what actions can be taken with Zope objects. Just as roles abstract users, permissions abstract objects. For example, many Zope objects, including DTML Methods and DTML Documents, can be viewed. This action is protected by the *View* permission.

Some permissions are only relevant for one type of object, for example, the *Change DTML Methods* permission only protects DTML Methods. Other permissions protect many types of objects, such as the *FTP access* and *WebDAV access* permissions which control whether objects are available via FTP and WebDAV.

You can find out what permissions are available on a given object by going to the *Security* management tab.



Acquire permission settings?	Permission	Anonymous	Manager	Owner
✓	Access contents information	☐	✓	☐
✓	Change configuration	☐	✓	☐
✓	Change permissions	☐	✓	☐
✓	Delete objects	☐	✓	☐
✓	Manage properties	☐	✓	☐
✓	Take ownership	☐	✓	☐
✓	Undo changes	☐	✓	☐
✓	Use mailhost services	☐	✓	☐
✓	View	☐	✓	☐
✓	View management screens	☐	✓	☐

Save Changes

You can define new roles by entering a role name and clicking the "Add Role" button.

User defined roles

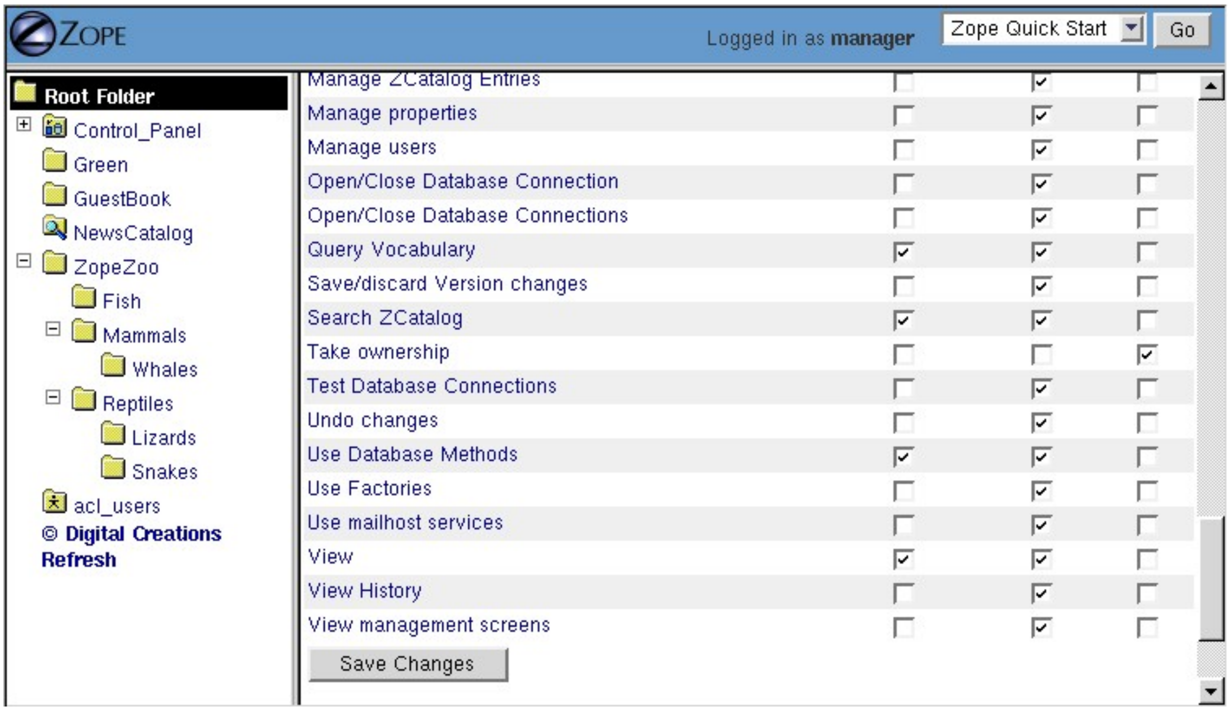
Figure 6−3 Security settings for a mail host object.

As you can see in Figure 6−3, a mail host has a limited pallet of permissions available. Contrast this to the many permissions that you see when setting security on a folder.

Defining Security Policies

Security policies are where roles meet permissions. Security policies define who can do what in a given part of the site.

You can set security policies on almost any Zope object. To set a security policy, go the *Security* tab. For example, click on the security tab of the root folder.



Manage ZCatalog Entries	☐	☑	☐
Manage properties	☐	☑	☐
Manage users	☐	☑	☐
Open/Close Database Connection	☐	☑	☐
Open/Close Database Connections	☐	☑	☐
Query Vocabulary	☑	☑	☐
Save/discard Version changes	☐	☑	☐
Search ZCatalog	☑	☑	☐
Take ownership	☐	☐	☑
Test Database Connections	☐	☑	☐
Undo changes	☐	☑	☐
Use Database Methods	☑	☑	☐
Use Factories	☐	☑	☐
Use mailhost services	☐	☑	☐
View	☑	☑	☐
View History	☐	☑	☐
View management screens	☐	☑	☐

Figure 6−4 Security policy for the root folder.

There is a lot going on in Figure 6−4. In the center of the screen is a grid of check boxes. The vertical columns of the grid represent roles, and the horizontal rows of the grid represent permissions. Checking the box at the intersection of a permission and a role grants users with that role the ability to take actions protected by that permission.

You'll notice that Zope comes with a default security policy that allows managers to perform most tasks, and anonymous users to perform only a couple. You can tailor this policy to suit your needs, by changing the security settings in the root folder.

For example, you can make your site private by disallowing anonymous users the ability to view any web pages. To do this deny all anonymous users View access by unchecking the *View* Permission where it intersects the *Anonymous* role. You can make your entire site private by making this security policy change in the root folder. If you want to make one part of your site private, you could make this change in the folder you want to make private.

This example points out a very important point about security policies: they control security for a given part of the site only. The only global security policy is the one on the root folder.

Security Policy Acquisition

How do different security policies interact? We've seen that you can create security policies on different objects, but what determines which policies control which objects? The answer is that objects use their own policy if they have one, additionally they acquire their parents' security policies through a process called *acquisition*.

Acquisition is a mechanism in Zope for sharing information among objects contained in a folder and its subfolders. The Zope security system uses acquisition to share security policies so that access can be controlled from high−level folders.

You can control security policy acquisition from the *Security* tab. Notice that there is a column of check boxes to the left of the screen labeled *Acquire permission settings*. Every check box in this column is checked by default. This means that security policy will acquire its parent's setting for each permission to role setting in addition to any settings specified on this screen. Keep in mind that for the root folder (which has no parent to acquire from) this left most check box column does not exist.

So for example, suppose you want to make this folder private. As we saw before this merely requires denying the *Anonymous* role the *View* permission. But as you can see on this screen, the Anonymous role doesn't have the View permission, and yet this folder is not private. Why is this? The answer is that the *Acquire permission settings* option is checked for the View permission. This means that the current settings are augmented by the security policies of this folder's parents. Somewhere above this folder the *Anonymous* role must be assigned to the *View* permission. You can verify this by examining the security policies of this folder's parents. To make the folder private we must uncheck the *Acquire permission settings* option. This will ensure that only the settings explicitly in this security policy are in effect.

In general, you should always acquire security settings unless you have a specific reason not too. This will make managing your security settings much easier as much of the work can be done from the root folder.

Next we'll consider some examples of how to create effective security policies using the tools that you've learned about so far in this chapter.

Security Usage Patterns

The basic concepts of Zope security are simple: roles and permissions combine to create security policies, and users actions are controlled by these policies. However these simple tools can be put together in many different ways. This can make managing security complex. Let's look at some basic patterns for managing security that provide good examples of how to create an effective and easy to manage security architecture.

Security Rules of Thumb

Here are a few simple guidelines for Zope security management. The security patterns that follow offer more specific recipes, but these guidelines give you some guidance when you face uncharted territory.

1. Define users at their highest level of control, but no higher.
2. Group objects that should be managed by the same people together in folders.
3. Keep it simple.

Rules one and two are closely related. Both are part of a more general rule for Zope site architecture. In general you should refactor your site to locate related resources and users near each other. Granted it's never possible to force resources and users into a strict hierarchy. However, a well considered arrangement of resources and users into folders and sub−folders helps tremendously.

Regardless of your site architecture, try to keep things simple. The more you complicate your security settings the harder time you'll have understanding it, managing it and making sure that it's effective. For example, limit the number of new roles you create, and try to use security policy acquisition to limit the number of places you have to explicitly define security settings. If you find that your security policies, users, and roles are growing into a complex thicket, you should rethink what you're doing; there's probably a simpler way.

Global and Local Policies

The most basic Zope security pattern is to define a global security policy on the root folder and acquire this policy everywhere. Then as needed you can add additional policies deeper in the object hierarchy to augment the global policy. Try to limit the number of places that you override the global policy. If you find that you have to make changes in a number of places, consider consolidating the objects in those separate locations into the same folder so that you can make the security settings in one place.

You should choose to acquire permission settings in your sub−policies unless your sub−policy is more restrictive than the global policy. In this case you should uncheck this option for the permission that you want to restrict.

This simple pattern will take care of much of your security needs. Its advantages are that it is easy to manage and easy to understand. These are extremely important characteristics for any security architecture.

Delegating Control to Local Managers

This security pattern is very central to Zope, and is part of what gives Zope its unique flavor. Zope encourages you to collect like resources in folders together and then to create user accounts in these folders to manager their contents.

Lets say you want to delegate the management of the *Sales* folder in your Zope site over to the new sales web manager, Steve. First, you don't want Steve messing with anything other than the Sales folder, so you don't need to add him to the acl_users folder in the root folder. Instead, create a new user folder in the *Sales* folder.

Now you can add Steve to the user folder in *Sales* and give him the Role *Manager*. Steve can now log directly into the Sales folder to manage his area of control by pointing his browser to *http://www.zopezoo.org/Sales/manage*.

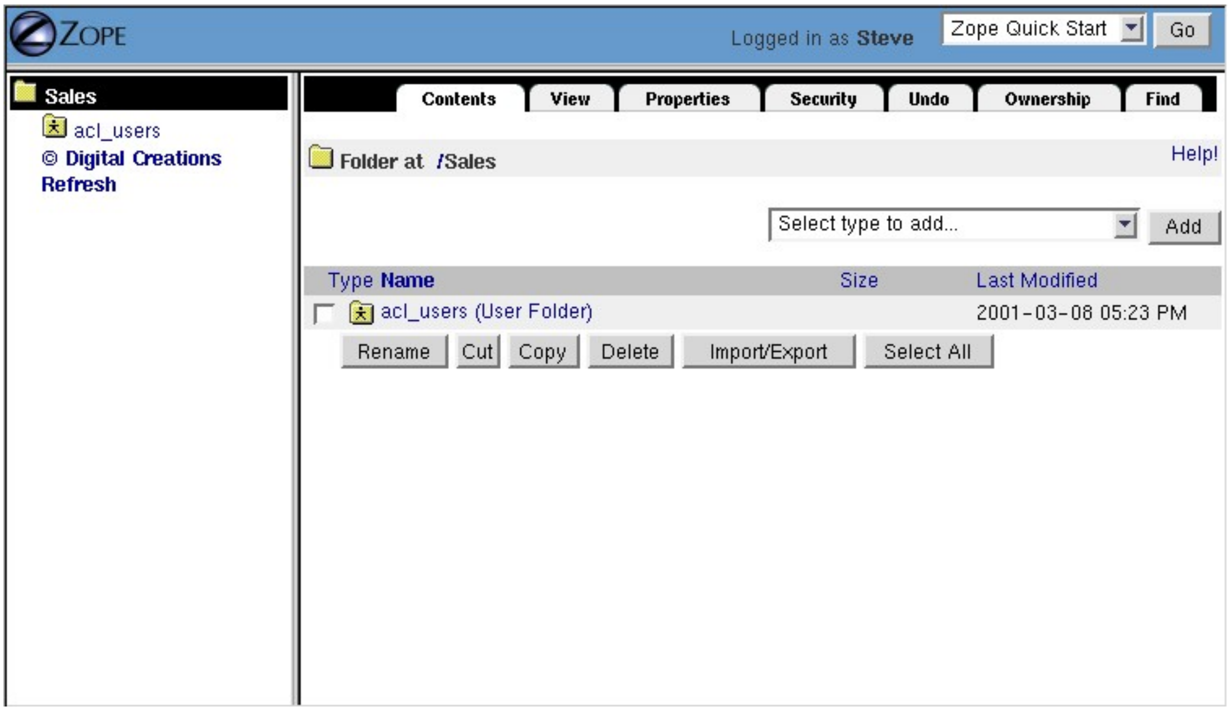


Figure 6−5 Managing the Sales folder.

Notice in Figure 6−5, the navigation tree on the left shows that *Sales* is the root folder. The local manager defined in this folder will never have the ability to log into any folders above *Sales* so it is shown as the top folder.

This pattern is very powerful since it can be applied recursively. For example, Steve can create a sub−folder for multi−level marketing sales. Then he can create a user folder in the multi−level marketing sales folder to delegate control of this folder to the multi−level marketing sales manager. And so on. This is a recipe for *huge* web sites managed by thousands of people.

The beauty of this pattern is that higher level managers need not concern themselves too much with what their underlings do. If they choose they can pay close attention, but they can safely ignore the details since they know that their delegates cannot make any changes outside their area of control, and they know that their security settings will be acquired.

Different Levels of Access with Roles

The local manager pattern is powerful and scalable, but it takes a rather coarse view of security. Either you have access or you don't. Sometimes you need to have more fine grained control. Many times you will have resources that need to be used by more than one type of person. Roles provides you with a solution to this problem. Roles allow you to define classes of users and set security policies for them.

Before creating new roles make sure that you really need them. Suppose that you have a web site that publishes articles. The public reads articles and managers edit and publish articles, but there is a third class of user who can author articles, but not publish or edit them.

One solution would be to create an authors folder where author accounts are created and given the *Manager* role. This folder would be private so it could only be viewed by managers. Articles could be written in this folder and then managers could move the articles out of this folder to publish them. This is a reasonable solution, but it requires that authors work only in one part of the site and it requires extra work by managers to move articles out of the authors folder. Also, consider that problems that result when an author wants to update an article that has been moved out of the authors folder.

A better solution is to add an *Author* role. Adding a role helps us because it allows access controls not based on location. So in our example, by adding an author role we make it possible for articles to be written, edited, and published anywhere in the site. We can set a global security policy that gives authors the ability to create and write articles, but doesn't grant them permissions to publish or edit articles.

Roles allow you to control access based on who a user is, not just where they are defined.

Controlling Access to Locations with Roles

Roles can help you overcome another subtle problem with the local manager pattern. The problem is that the local manager pattern requires a strict hierarchy of control. There is no provision to allow two different groups of people to access the same resources without one group being the manager of the other group. Put another way, there is no way for users defined in one part of the site to manage resources in another part of the site.

Let's take an example to illustrate the second limitation of the local manager pattern. Suppose you run a large site for a pharmaceutical company. You have two classes of users, scientists and salespeople. In general the scientists and the salespeople manage different web resources. However, suppose that there are some things that both types of people need to manage, such as advertisements that have to contain complex scientific warnings. If we define our scientists in the *Science* folder and the salespeople in the *Sales* folder, where should we put the *AdsWithComplexWarnings* folder? Unless the Science folder is inside the Sales folder or vice versa there is no place that we can put the *AdsWithComplexWarnings* folder so that both scientists and salespeople can manage it. It is not a good political or practical solution to have the salespeople manage the scientists or vice versa; what can be done?

The solution is to use roles. You should create two roles at a level above both the Science and Sales folders, say *Scientist*, and *SalesPerson*. Then instead of defining the scientists and salespeople in their own folders define them higher in the object hierarchy so that they have access to the *AdsWithComplexWarnings* folder.

When you create users at this higher level, you should not give them the *Manager* role, but instead give them Scientist or SalesPerson as appropriate. Then you should set the security policies. On the *Science* folder the *Scientist* role should have the equivalent of *Manager* control. On the *Sales* folder, the *Salesperson* role should have the same permissions as *Manager*. Finally on the *AdsWithComplexWarnings* folder you should give both *Scientist* and *Salesperson* roles adequate permissions. This way roles are used not to provide different levels of access, but to provide access to different locations based on who you are.

Another common situation when you might want to employ this pattern is when you cannot define your managers locally. For example, you may be using an alternate user folder that requires all users to be defined in the root folder. In this case you would want to make extensive use of roles to limit access to different locations based on roles.

This wraps up our discussion of security patterns. By now you should have a reasonable grasp of how to use user folders, roles, and security policies, to shape a reasonable security architecture for your application. Next we'll cover two advanced security issues, how to perform security checks, and securing executable content.

Performing Security Checks

Most of the time you don't have to perform any security checks. If a user attempts to perform a secured operation, Zope will prompt them to log in. If the user doesn't have adequate permissions to access a protected resource, Zope will deny them access. However, sometimes you may wish to manually perform security checks. The main reason to do this is to limit the choices you offer a user to those for which they are authorized. This doesn't prevent a sneaky user for trying to access secured actions, but it does reduce user frustration, by not giving to user the option to try something that will not work.

The most common security query asks whether the current user has a given permission. For example, suppose your application allows some users to upload files. This action may be protected by the "Add Documents, Images, and Files" standard Zope permission. You can test to see if the current user has this permission in DTML:

```
<dtml-if expr="_.SecurityCheckPermission(
              'Add Documents, Images, and Files', this())">

  <form action="upload">
  ...
  </form>

</dtml-if>
```

The *SecurityCheckPermission* function takes two arguments, a permission name, and an object. In this case we pass `this()` as the object which is a reference to the current object. By passing the current object, we make sure that local roles are taken into account when testing whether the current user has a given permission.

You can find out about the current user by accessing the user in DTML. The current user is a Zope object like any other and you can perform actions on it using methods defined in the API documentation.

Suppose you wish to display the current user name on a web page to personalize the page. You can do this easily in DTML:

```
<dtml-var expr="_.SecurityGetUser().getUserName()">
```

You can retrieve the currently logged in user with the *SecurityGetUser* DTML function. This DTML fragment tests the current user by calling the *getUserName* method on the current user object. If the user is not logged in, you will get the name of the anonymous user, which is *Anonymous User*.

Next we'll look at another advanced issue which affects security of DTML and scripts.

Advanced Security Issues: Ownership and Executable Content

You've now covered all the basics of Zope security. What remains are the advanced concepts of *ownership* and *executable content*. Zope uses ownership to associate objects with users who create them, and executable content refers to objects such as Scripts, DTML Methods and Documents, which execute user code.

For small sites with trusted users you can safely ignore these advanced issues. However for large sites where you allow untrusted users to create and manage Zope objects, it's important to understand ownership and securing executable content.

The Problem: Trojan Horse Attacks

The basic scenario that motivates both ownership and executable content controls is a *Trojan horse* attack. A Trojan horse is an attack on a system that operates by tricking a user into taking a potentially harmful action. A typical Trojan horse masquerades as a benign program that causes harm when you unwittingly run it.

All web−based platform including Zope and many others are subject to this style of attack. All that is required is to trick someone into visiting a URL that performs a harmful action.

This kind of attack is very hard to protect against. You can trick someone into clicking a link fairly easily, or you can use more advanced techniques such as Javascript to cause a user to visit a malicious URL.

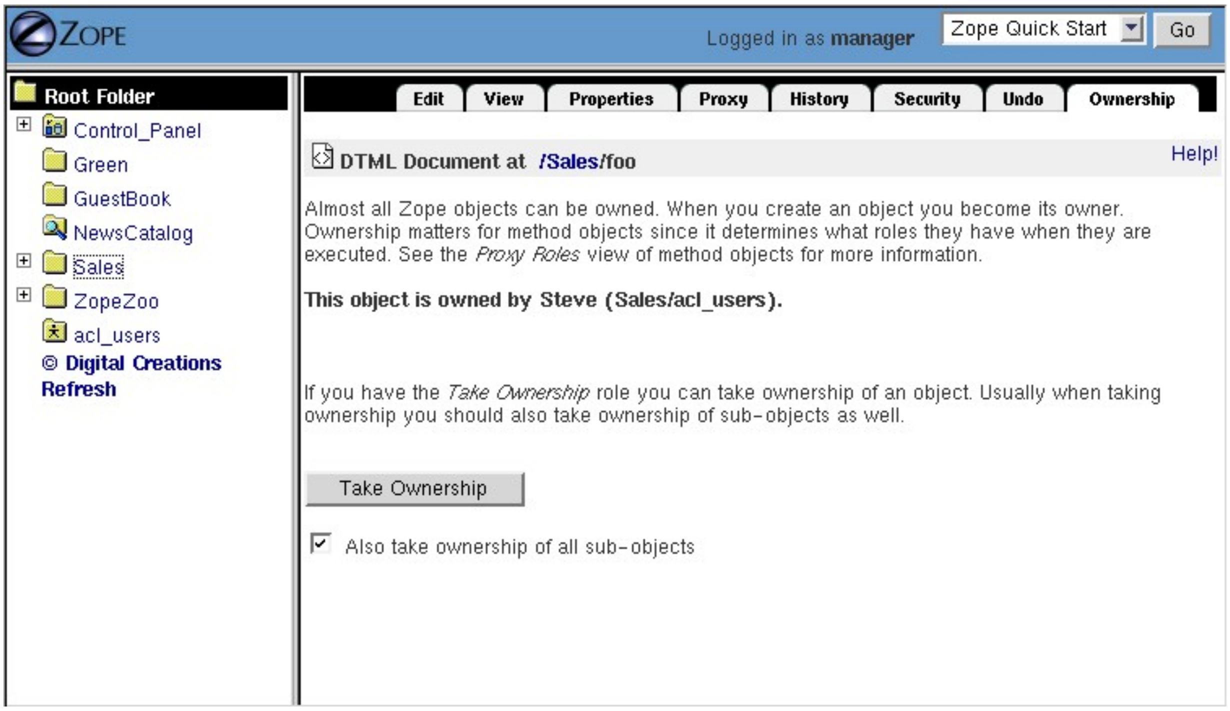
Zope offers some protection from this kind of Trojan horse. Zope helps protect your site from server−side to Trojan attacks by limiting the power of web resources based on who authored them. If an untrusted user author a web page, then the power of the web pages to do harm to unsuspecting visitors will be limited. For example, suppose an untrusted user creates a DTML document or Python script that deletes all the pages in your site. If they attempt to view the page, it will fail since they do not have adequate permissions. If a manager views the page, it will also fail, even though the manager does have adequate permissions to perform the dangerous action.

Zope uses ownership information and executable content controls to provide this limited protection.

Managing Ownership

When a user creates a Zope object, they *own* that object. An object that has no owner is referred to as *unowned*. Ownership information is stored in the object itself. This is similar to how UNIX keeps track of the owner of a file.

You find out how an object is owned by viewing the *Ownership* management tab, as shown in Figure 6−6.



DTML Document at /Sales/foo

Almost all Zope objects can be owned. When you create an object you become its owner. Ownership matters for method objects since it determines what roles they have when they are executed. See the *Proxy Roles* view of method objects for more information.

This object is owned by Steve (Sales/acl_users).

If you have the *Take Ownership* role you can take ownership of an object. Usually when taking ownership you should also take ownership of sub-objects as well.

☑ Also take ownership of all sub-objects

Figure 6−6 Managing ownership settings.

This screen tells you if the object is owned and if so by whom. If the object is owned by someone else, and you have the *Take ownership* permission, you can take over the ownership of an object. You also have the option of taking ownership of all sub−objects by checking the *Take ownership of all sub−objects* box. Taking ownership is mostly useful if the owner account has been deleted, or if objects have been turned over to you for continued management.

As we mentioned earlier in the chapter ownership affects security policies because a user will have the local role *Owner* on objects they own. However, ownership also affects security because it controls the role's executable content.

Roles of Executable Content

Through the web you can edit scripts on some kinds of Zope objects. These objects including DTML Documents, DTML Methods, SQL Methods, Python−based Scripts, and Perl−based Scripts. These objects are said to be *executable* since they run scripts that are edited through the web.

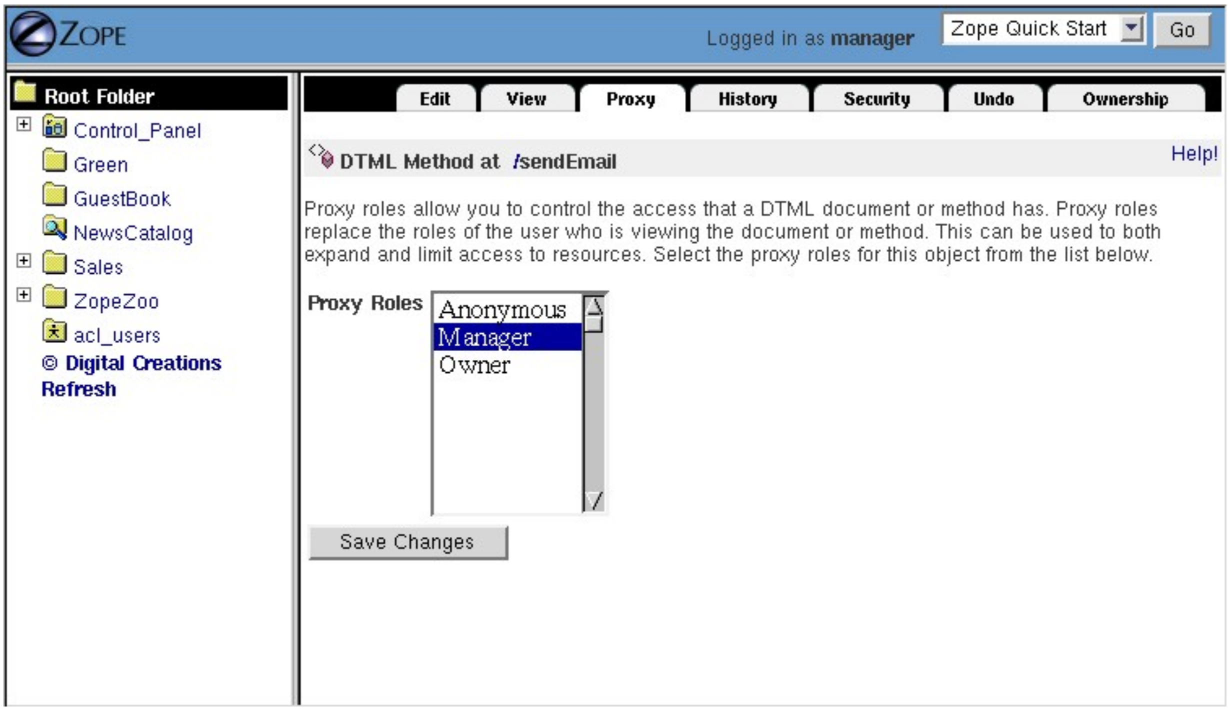
When you visit an executable object by going to its URL or calling it from DTML or a script, Zope runs the object's script. The script is restricted by the roles of the object's owner and your roles. In other words an executable object can only perform actions that *both* the owner and the viewer are authorized for. This keeps an unprivileged user from writing a harmful script and then tricking a powerful user into executing the script. You can't fool someone else into performing an action that you are not authorized to perform yourself. This is how Zope uses ownership is used to protect against server−side Trojan horse attacks.

Proxy Roles

Sometimes Zope's system of limiting access to executable objects isn't exactly what you want. Sometimes you may wish to clamp down security on an executable object despite whoever may own or execute it as a form of extra security. Other times you may want to provide an executable object with extra access to allow an unprivileged viewer to perform protected actions. *Proxy roles* provide you with a way to tailor the roles of an executable object.

Suppose you want to create a mail form that allows anonymous users to send email to the webmaster of your site. Sending email is protected by the `Use mailhost services` permission. Anonymous users don't normally have this permission and for good reason. You don't want just anyone to be able to anonymously send email with your Zope server.

The problem with this arrangement is that your DTML Method that sends email will fail for anonymous users. How can you get around this problem? The answer is to set the proxy roles on the DTML Method that sends email so that when it executes it has the "Manager" role. Visit the Proxy management tab on your DTML Method, as shown in Figure 6−7.



DTML Method at /sendEmail

Proxy roles allow you to control the access that a DTML document or method has. Proxy roles replace the roles of the user who is viewing the document or method. This can be used to both expand and limit access to resources. Select the proxy roles for this object from the list below.

Proxy Roles: Manager

Figure 6−7 Proxy role management.

Select *Manager* and click the *Change* button. This will set the proxy roles of the mail sending method to *Manager*. Note you must have the *Manager* role yourself to set it as a proxy role. Now when anyone, anonymous or not runs your mail sending method, it will execute with the *Manager* role, and thus will have authorization to send email.

Proxy roles define a fixed set of the permissions of executable content. Thus you can also use them to restrict security. For example, if you set the proxy roles of a script to *Anonymous* role, then the script will never execute any other roles besides *Anonymous* despite the roles of the owner and viewer.

Use Proxy roles with care, since they can be used skirt the default security restrictions.

Summary

Security consists of two processes, authentication and authorization. User folders control authentication, and security policies control authorization. Zope security is intimately tied with the concept of location; users have location, security policies have location, even roles can have location. Creating an effective security architecture requires attention to location. When in doubt refer to the security usage patterns discussed in this chapter.

In the next chapter we'll switch gears and explore advanced DTML. DTML can be a very powerful tool for presentation and scripting. You'll find out about many new tags, and will take a look at some DTML−specific security controls that were not covered in this chapter.

Chapter 8: Variables and Advanced DTML

DTML is the kind of language that "does what you mean." That is good, when it does what you actually want it to do, but when it does something you don't want to do, it's bad. This chapter tells you how to make DTML do what you *really* mean.

It's no lie that DTML has reputation for complexity. And it's true, DTML is really simple if you all you want to do is simple layout, like you've seen so far. However, if you want to use DTML for more advanced tasks, you have to understand where DTML variables come from.

Here's a very tricky error that almost all newbies encounter. Imagine you have a DTML Document named called *zooName*. This document contains an HTML form like the following:

```
<dtml-var standard_html_header>

  <dtml-if zooName>

    <p><dtml-var zooName></p>

  <dtml-else>

    <form action="<dtml-var URL>" method="GET">
      <input name="zooName">
      <input type="submit" value="What is zooName?">
    </form>

  </dtml-if>

<dtml-var standard_html_footer>
```

This looks simple enough, the idea is, this is an HTML page that calls itself. This is because the HTML action is the *URL* variable, which will become the URL of the DTML Document.

If there is a `zooName` variable, then the page will print it, if there isn't, it shows a form that asks for it. When you click submit, the data you enter will make the "if" evaluate to true, and this code should print what entered in the form.

But unfortunately, this is one of those instances where DTML will not do what you mean, because the name of the DTML Document that contains this DTML is also named *zooName*, and it doesn't use the variable out of the request, it uses itself, which causes it call itself and call itself, ad infinitum, until you get an "excessive recursion" error. So instead of doing what you really meant, you got an error. This is what confuses beginners. In the next couple sections, we'll show you how to fix this example to do what you mean.

How Variables are Looked up

There's are actually two ways to fix the DTML error in the *zooName* document. The first is that you can rename the document to something like *zopeNameFormOrReply* and always remember this special exception and never do it; never knowning why it happens. The second is to understand how names are looked up, and to be explicit about where you want the name to come from in the *namespace*.

The DTML namespace is a collection of objects arranged in a *stack*. A stack is a list of objects that can manipulated by *pushing* and *popping* objects on to and off of the stack.

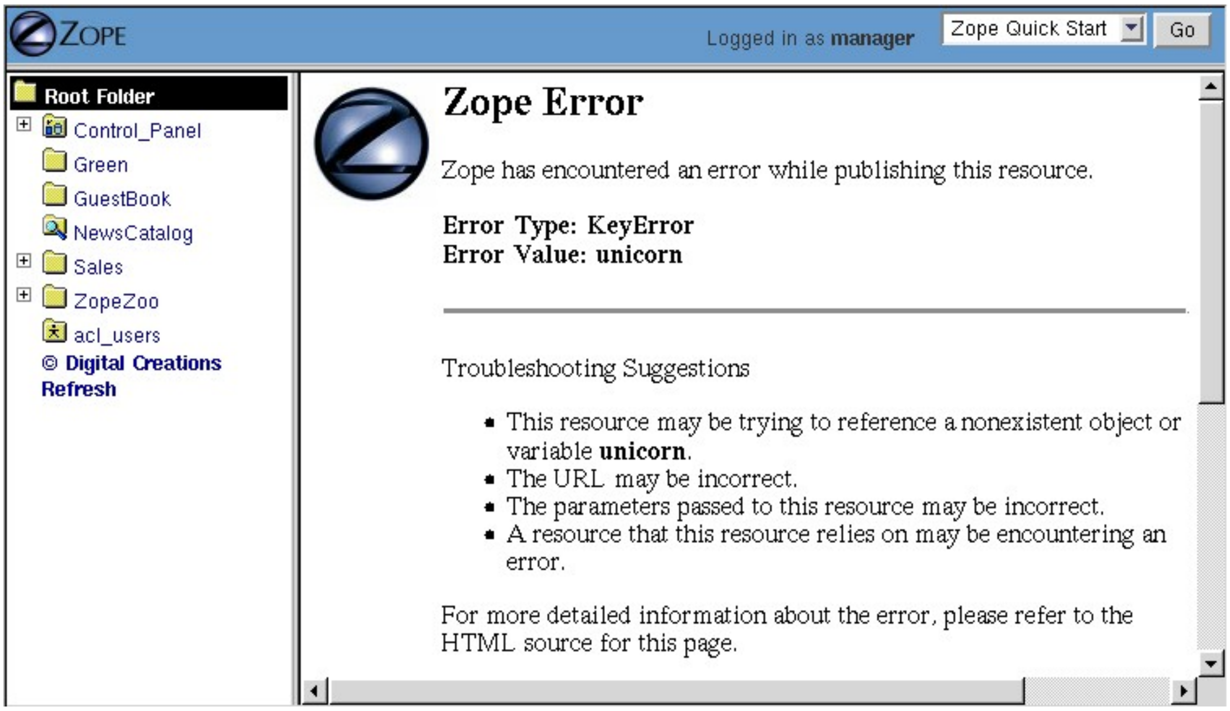
When a DTML Document or DTML Method is executed, Zope creates a DTML namespace to resolve DTML variable names. It's important to understand the workings of the DTML namespace so that you can accurately predict how Zope will locate variables. Some of the trickiest problems you will run into with DTML can be resolved by understanding the DTML namespace.

When Zope looks for names in the DTML namespace stack it first looks at the very top most object in the stack. If the name can't be found there, then the next item down is looked in. Zope will work its way down the stack, checking each object in turn until it finds the name that it is looking for.

If Zope gets all the way down to the bottom of the stack and can't find what it is looking for, then an error is generated. For example, try looking for the non−existent name, *unicorn*:

```
<dtml-var unicorn>
```

As long as there is no variable named *unicorn* viewing this DTML will return an error, as shown in Figure 7−1.



Zope Error

Zope has encountered an error while publishing this resource.

Error Type: KeyError
Error Value: unicorn

Troubleshooting Suggestions

- This resource may be trying to reference a nonexistent object or variable **unicorn**.
- The URL may be incorrect.
- The parameters passed to this resource may be incorrect.
- A resource that this resource relies on may be encountering an error.

For more detailed information about the error, please refer to the HTML source for this page.

Figure 7−1 DTML error message indicating that it cannot find a variable.

But the DTML stack is not all there is to names because DTML doesn't start with an empty stack, before you even begin executing DTML in Zope there are already a number of objects pushed on the namespace stack.

DTML Namespaces

DTML namespaces are built dynamically for every request in Zope. When you call a DTML Method or DTML Document through the web, the DTML namespace starts with the same first two stack elements the client object and the request as shown in Figure 7−2

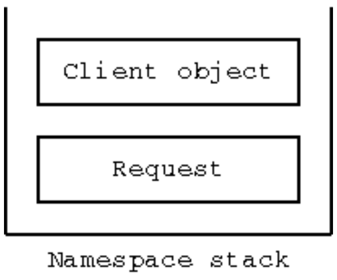


Figure 7−2 Initial DTML namespace stack.

The client object is the first object on the top of the DTML namespace stack. What the client object is depends on whether or not you are executing a DTML Method or a DTML Document. In our example above, this means that the client object is named *zooName*. Which is why it breaks. The form input that we really wanted comes from the web request, but the client is looked at first.

The request namespace is always on the bottom of the DTML namespace stack, and is therefore the last namespace to be looked in for names. This means that we must be explicit in our example about which namespace we want. We can do this with the DTML `with` tag:

```
<dtml-var standard_html_header>

  <dtml-with REQUEST only>
    <dtml-if zooName>
      <p><dtml-var zooName></p>
    <dtml-else>
      <form action="<dtml-var URL>" method="GET">
        <input name="zooName">
        <input type="submit" value="What is zooName?">
      </form>
    </dtml-if>
  </dtml-with>

<dtml-var standard_html_footer>
```

Here, the with tag says to look in the `REQUEST` namespace, and *only* the `REQUEST` namespace, for the name "zooName".

DTML Client Object

The client object in DTML depends on whether or not you are executing a DTML Method or a DTML Document. In the case of a Document, the client object is always the document itself, or in other words, a DTML Document is its own client object.

A DTML Method however can have different kinds of client objects depending on how it is called. For example, if you had a DTML Method that displayed all of the contents of a folder then the client object would be the folder that is being displayed. This client object can change depending on which folder the method in question is displaying. For example, consider the following DTML Method named *list* in the root folder:

```
<dtml-var standard_html_header>

<ul>
<dtml-in objectValues>
  <li><dtml-var title_or_id></li>
</dtml-in>
</ul>
```

```
        <dtml-var standard_html_footer>
```

Now, what this method displays depends upon how it is used. If you apply this method to the *Reptiles* folder with the URL `http://localhost:8080/Reptiles/list`, then you will get something that looks like Figure 7−3.

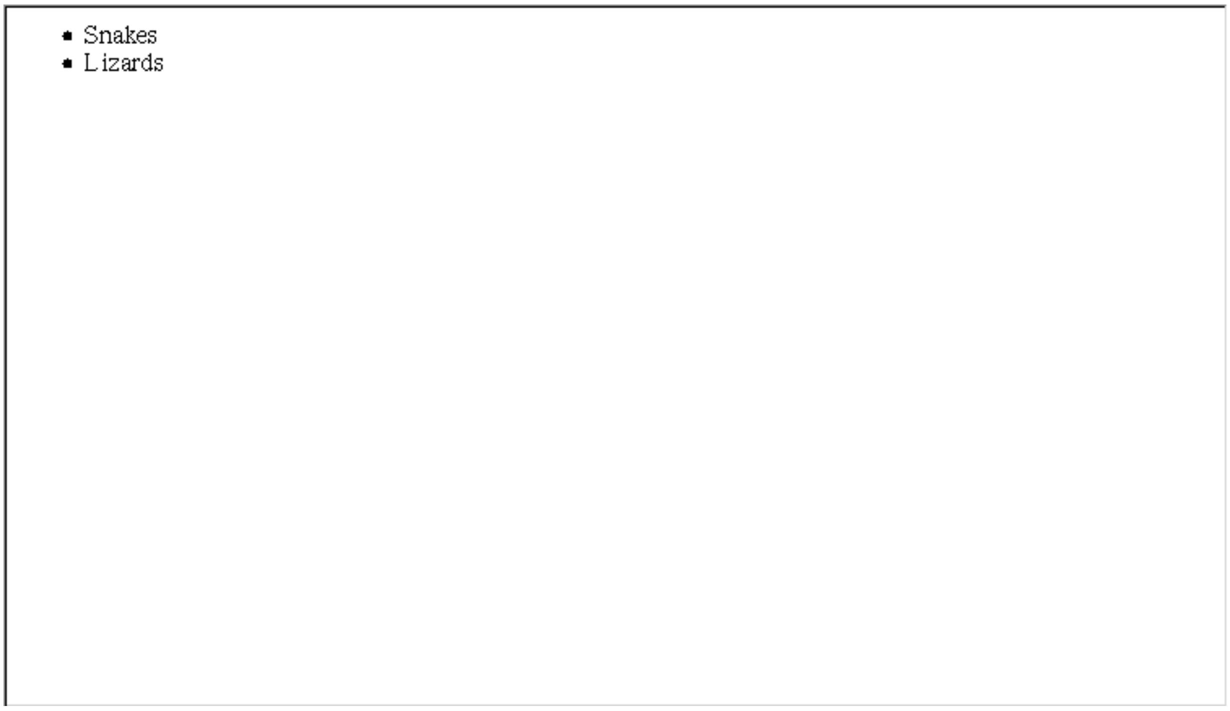

- Snakes
- Lizards

Figure 7−3 Applying the list method to the Reptiles folder.

But if you were to apply the method to the *Birds* folder with the URL *http://localhost:8080/Birds/list* then you would get something different, only two items in the list, *Parrot* and *Raptors*.

Same DTML Method, different results. In the first example, the client object of the *list* method was the *Reptiles* folder. In the second example, the client object was the *Birds* folder. When Zope looked up the *objectValues* variable, in the first case it called the *objectValues* method of the *Reptiles* folder, in the second case it called the *objectValues* method of the *Birds* folder.

In other words, the client object is where variables such as methods, and properties are looked up first.

As you saw in Chapter 4, "Dynamic Content with DTML", if Zope cannot find a variable in the client object, it searches through the object's containers. Zope uses acquisition to automatically inherit variables from the client object's containers. So when Zope walks up the object hierarchy looking for variables it always starts at the client object, and works its way up from there.

DTML Request Object

The request object is the very bottom most object on the DTML namespace stack. The request contains all of the information specific to the current web request.

Just as the client object uses acquisition to look in a number of places for variables, so too the request looks up variables in a number of places. When the request looks for a variable it consults these sources in order:

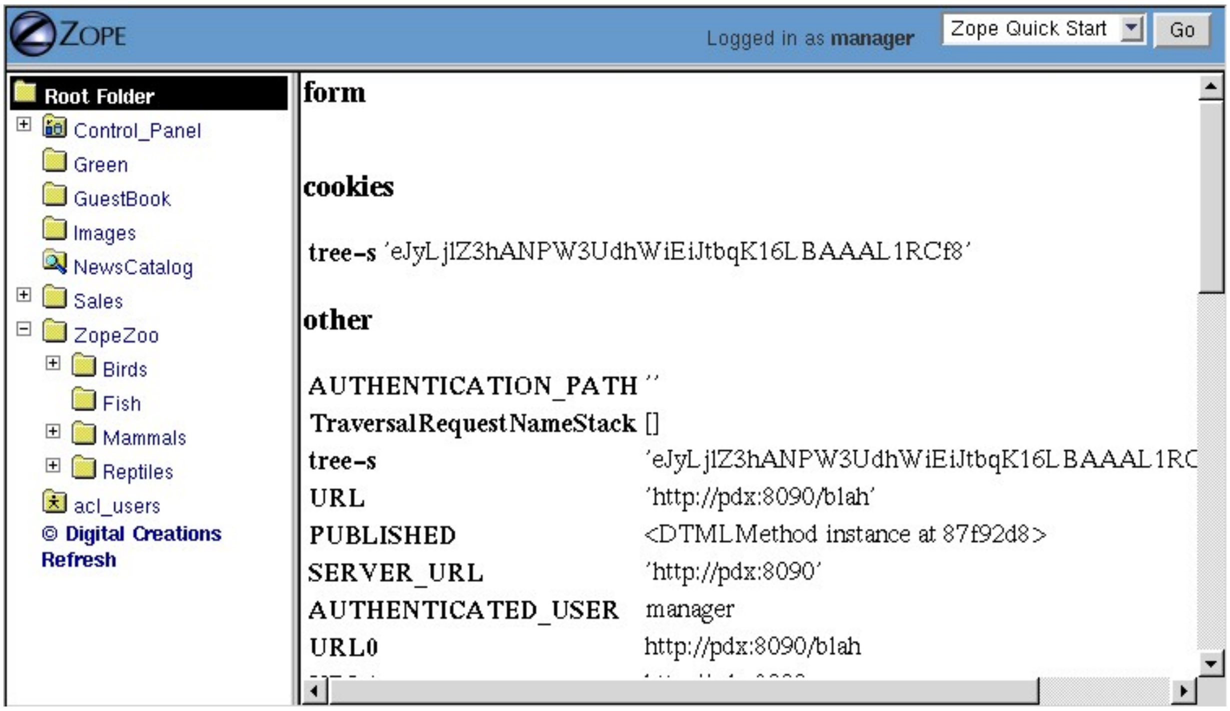
1. The CGI environment. The Common Gateway Interface, or CGI interface defines a standard set of environment variables to be used by dynamic web scripts. These variables are provided by Zope in the REQUEST namespace.
2. Form data. If the current request is a form action, then any form input data that was submitted with the request can be found in the REQUEST object.
3. Cookies. If the client of the current request has any cookies these can be found in the current REQUEST object.
4. Additional variables. The REQUEST namespace provides you with lots of other useful information, such as the URL of the current object and all of its parents.

The request namespace is very useful in Zope since it is the primary way that clients (in this case, web browsers) communicate with Zope by providing form data, cookies and other information about themselves. For more information about the request object, see Appendix B.

A very simple and enlightening example is to simply print the REQUEST out in an HTML page:

```
<dtml-var standard_html_header>

<dtml-var REQUEST>

<dtml-var standard_html_footer>
```

Try this yourself, you should get something that looks like Figure 7−4.



form

cookies

```
tree-s 'eJyLjlZ3hANPW3UdhWiEiJtbqK16LBAAAL1RCf8'
```

other

```
AUTHENTICATION_PATH ''
TraversalRequestNameStack []
tree-s                'eJyLjlZ3hANPW3UdhWiEiJtbqK16LBAAAL1RC
URL                   'http://pdx:8090/blah'
PUBLISHED             <DTMLMethod instance at 87f92d8>
SERVER_URL            'http://pdx:8090'
AUTHENTICATED_USER    manager
URL0                  http://pdx:8090/blah
```

Figure 7−4 Displaying the request.

Since the request comes after the client object, if there are names that exist in both the request and the client object, DTML will always find them first in the client object. This can be a problem. Next, let's look at some ways to get around this problem by controlling more directly how DTML looks up variables.

Rendering Variables

When you insert a variable using the *var* tag, Zope first looks up the variable using the DTML namespace, it then *renders* it and inserts the results. Rendering means turning an object or value into a string suitable for inserting into the output. Zope renders simple variables by using Python's standard method for coercing objects to strings. For complex objects such as DTML Methods and SQL Methods, Zope will call the object instead of just trying to turn it into a string. This allows you to insert DTML Methods into other DTML Methods.

In general Zope renders variables in the way you would expect. It's only when you start doing more advanced tricks that you become aware of the rendering process. Later in this chapter we'll look at some examples of how to control rendering using the `getitem` DTML utility function.

Modifying the DTML Namespace

Now that you have seen that the DTML namespace is a stack, you may be wondering how, or even why, new objects get pushed onto it.

Some DTML tags modify the DTML namespace while they are executing. A tag may push some object onto the namespace stack during the course of execution. These tags include the *in* tag, the *with* tag, and the *let* tag.

In Tag Namespace Modifications

When the *in* tag iterates over a sequence it pushes the current item in the sequence onto the top of the namespace stack:

```
<dtml-var getId> <!-- This is the id of the client object -->

<dtml-in objectValues>

  <dtml-var getId> <!-- this is the id of the current item in the
                     objectValues sequence -->
</dtml-in>
```

You've seen this many times throughout the examples in this book. While the *in* tag is iterating over a sequence, each item is pushed onto the namespace stack for the duration of the contents of the in tag block. When the block is finished executing, the current item in the sequence is popped off the DTML namespace stack and the next item in the sequence is pushed on.

The *With* Tag

The *with* tag pushes an object that you specify onto the top of the namespace stack for the duration of the with block. This allows you to specify where variables should be looked up first. When the with block closes, the object is popped off the namespace stack.

Consider a folder that contains a bunch of methods and properties that you are interested in. You could access those names with Python expressions like this:

```
<dtml-var standard_html_header>

<dtml-var expr="Reptiles.getReptileInfo()">
<dtml-var expr="Reptiles.reptileHouseMaintainer">

<dtml-in expr="Reptiles.getReptiles()">
```

```
  <dtml-var species>
</dtml-in>

<dtml-var standard_html_footer>
```

Notice that a lot of complexity is added to the code just to get things out of the *Reptiles* folder. Using the *with* tag you can make this example much easier to read:

```
<dtml-var standard_html_header>

<dtml-with Reptiles>

  <dtml-var getReptileInfo>
  <dtml-var reptileHouseMaintainer>

  <dtml-in getReptiles>
    <dtml-var species>
  </dtml-in>

</dtml-with>

<dtml-var standard_html_footer>
```

Another reason you might want to use the *with* tag is to put the request, or some part of the request on top of the namespace stack. For example suppose you have a form that includes an input named *id*. If you try to process this form by looking up the *id* variable like so:

```
<dtml-var id>
```

You will not get your form's id variable, but the client object's id. One solution is to push the web request's form on to the top of the DTML namespace stack using the *with* tag:

```
<dtml-with expr="REQUEST.form">
  <dtml-var id>
</dtml-with>
```

This will ensure that you get the form's id first. See Appendix B for complete API documentation of the request object.

If you submit your form without supplying a value for the *id* input, the form on top of the namespace stack will do you no good, since the form doesn't contain an *id* variable. You'll still get the client object's id since DTML will search the client object after failing to find the *id* variable in the form. The *with* tag has an attribute that lets you trim the DTML namespace to only include the object you specify:

```
<dtml-with expr="REQUEST.form" only>
  <dtml-if id>
    <dtml-var id>
  <dtml-else>
    <p>The form didn't contain an "id" variable.</p>
  </dtml-if>
</dtml-with>
```

Using the *only* attribute allows you to be sure about where your variables are being looked up.

The *Let* Tag

The *let* tag lets you push a new namespace onto the namespace stack. This namespace is defined by the tag attributes to the *let* tag:

```
<dtml-let person="'Bob'" relation="'uncle'">
  <p><dtml-var person>'s your <dtml-var relation>.</p>
</dtml-let>
```

This would display:

```
<p>Bob's your uncle.</p>
```

The *let* tag accomplishes much of the same goals as the *with* tag. The main advantage of the let tag is that you can use it to define multiple variables to be used in a block. The *let* tag creates one or more new variables and their values and pushes a namespace object containing those variables and their values on to the top of the DTML namespace stack. In general the *with* tag is more useful to push existing objects onto the namespace stack, while the *let* tag is better suited for defining new variables for a block.

When you find yourself writing complex DTML that requires things like new variables, there's a good chance that you could do the same thing better with Python or Perl. Advanced scripting is covered in Chapter 10, "Advanced Zope Scripting".

The DTML namespace is a complex place, and this complexity evolved over a lot of time. Although it helps to understand where names come from, it is much more helpful to always be specific about where you are looking for a name. The `with` and `let` tags let you control the namespace to look exactly in the right place for the name you are looking for.

DTML Namespace Utility Functions

Like all things in Zope, the DTML namespace is an object, and it can can be accessed directly in DTML with the _ (underscore) object. The _ namespace is often referred to as as "the under namespace".

The under namespace provides you with many useful methods for certain programming tasks. Let's look at a few of them.

Say you wanted to print your name three times. This can be done with the *in* tag, but how do you explicitly tell the *in* tag to loop three times? Just pass it a sequence with three items:

```
<dtml-var standard_html_header>

<ul>
<dtml-in expr="_.range(3)">
  <li><dtml-var sequence-item>: My name is Bob.</li>
</dtml-in>
</ul>

<dtml-var standard_html_footer>
```

The `_.range(3)` Python expression will return a sequence of the first three integers, 0, 1, and 2. The *range* function is a *standard Python built−in* and many of Python's built−in functions can be accessed through the _ namespace, including:

`range([start,], stop, [step])`

Returns a list of integers from `start` to `stop` counting `step` integers at a time. `start` defaults to 0 and `step` defaults to 1. For example:

```
'_.range(3,9,2)' -- gives '[3,5,7,9]'.

'len(sequence)' -- 'len' returns the size of *sequence* as an integer.
```

Many of these names come from the Python language, which contains a set of special functions called `built-ins`. The Python philosophy is to have a small, set number of built−in names. The Zope philosphy can be thought of as having a large, complex array of built−in names.

The under namespace can also be used to explicitly control variable look up. There is a very common usage of this syntax. You've seen that the in tag defines a number of special variables, like *sequence−item* and *sequence−key* that you can use inside a loop to help you display and control it. What if you wanted to use one of these variables inside a Python expression?:

```
<dtml-var standard_html_header>

<h1>The squares of the first three integers:</h1>
<ul>
<dtml-in expr="_.range(3)">
  <li>The square of <dtml-var sequence-item> is:
    <dtml-var expr="sequence-item * sequence-item">
  </li>
</dtml-in>
</ul>

<dtml-var standard_html_footer>
```

Try this, does it work? No! Why not? The problem lies in this var tag:

```
<dtml-var expr="sequence-item * sequence-item">
```

Remember, everything inside a Python expression attribute must be a *valid Python expression*. In DTML, *sequence−item* is the name of a variable, but in Python this means "The object *sequence* minus the object *item*". This is not what you want.

What you really want is to look up the variable *sequence−item*. One way to solve this problem is to use the *in* tag *prefix* attribute. For example:

```
<dtml-var standard_html_header>

<h1>The squares of the first three integers:</h1>
<ul>
<dtml-in prefix="loop" expr="_.range(3)">
  <li>The square of <dtml-var loop_item> is:
    <dtml-var expr="loop_item * loop_item">
  </li>
</dtml-in>
</ul>

<dtml-var standard_html_footer>
```

The *prefix* attribute causes *in* tag variables to be renamed using the specified prefix and underscores, rather than using "sequence" and dashes. So in this example, "sequence−item" becomes "loop_item". See Appendix A for more information on the *prefix* attribute.

Another way to look up the variable *sequence−item* in a DTML expression is to use the *getitem* utility function to explicitly look up a variable:

```
The square of <dtml-var sequence-item> is:
<dtml-var expr="_.getitem('sequence-item') *
                _.getitem('sequence-item')">
```

The *getitem* function takes the name to look up as its first argument. Now, the DTML Method will correctly display the sum of the first three integers. The *getitem* method takes an optional second argument which specifies whether or not to render the variable. Recall that rendering a DTML variable means turning it into a string. By default the *getitem* function does not render a variable.

Here's how to insert a rendered variable named *myDoc*:

```
<dtml−var expr="_.getitem('myDoc', 1)">
```

This example is in some ways rather pointless, since it's the functional equivalent to:

```
<dtml−var myDoc>
```

However, suppose you had a form in which a user got to select which document they wanted to see from a list of choices. Suppose the form had an input named *selectedDoc* which contained the name of the document. You could then display the rendered document like so:

```
<dtml−var expr="_.getitem(selectedDoc, 1)">
```

Notice in the above example that *selectedDoc* is not in quotes. We don't want to insert the variable named *selectedDoc* we want to insert the variable *named by selectedDoc*. For example, the value of *selectedDoc* might be *chapterOne*. Using indirect variable insertion you can insert the *chapterOne* variable. This way you can insert a variable whose name you don't know when you are authoring the DTML.

If you a python programmer and you begin using the more complex aspects of DTML, consider doing a lot of your work in Python scripts that you call *from* DTML. This is explained more in Chapter 10, "Advanced Zope Scripting". Using Python sidesteps many of the issues in DTML.

DTML Security

Zope can be used by many different kinds of users. For example, the Zope site, Zope.org, has over 11,000 community members at the time of this writing. Each member can log into Zope, add objects and news items, and manage their own personal area.

Because DTML is a scripting language, it is very flexible about working with objects and their properties. If there were no security system that constrained DTML then a user could potentially create malicious or privacy−invading DTML code.

DTML is restricted by standard Zope security settings. So if you don't have permission to access an object by going to its URL you also don't have permission to access it via DTML. You can't use DTML to trick the Zope security system.

For example, suppose you have a DTML Document named *Diary* which is private. Anonymous users can't access your diary via the web. If an anonymous user views DTML that tries to access your diary they will be denied:

```
<dtml−var Diary>
```

DTML verifies that the current user is authorized to access all DTML variables. If the user does not have authorization, than the security system will raise an *Unauthorized* error and the user will be asked to present more privileged authentication credentials.

In Chapter 7, "Users and Security" you read about security rules for executable content. There are ways to tailor the roles of a DTML Document or Method to allow it to access restricted variables regardless of the viewer's roles.

Safe Scripting Limits

DTML will not let you gobble up memory or execute infinite loops and recursions the restrictions on looping and memory are pretty tight, which makes DTML not the right language for complex, expensive programming logic. For example, you cannot create huge lists with the *_.range* utility function. You also have no way to access the filesystem directly in DTML.

Keep in mind however that these safety limits are simple and can be outsmarted by a determined user. It's generally not a good idea to let anyone you don't trust write DTML code on your site.

Advanced DTML Tags

In the rest of this chapter we'll look at the many advanced DTML tags. These tags are summarized in Appendix A. DTML has a set of built−in tags, as documented in this book, which can be counted on to be present in all Zope installations and perform the most common kinds of things. However, it is also possible to add new tags to a Zope installation. Instructions for doing this are provided at the Zope.org web site, along with an interesting set of contributed DTML tags.

This section covers what could be referred to as Zope *miscellaneous* tags. These tags don't really fit into any broad categories except for one group of tags, the *exception handling* DTML tags which are discussed at the end of this chapter.

The *Call* Tag

The *var* tag can call methods, but it also inserts the return value. Using the *call* tag you can call methods without inserting their return value into the output. This is useful if you are more interested in the effect of calling a method rather than its return value.

For example, when you want to change the value of a property, *animalName*, you are more interested in the effect of calling the *manage_changeProperties* method than the return value the method gives you. Here's an example:

```
<dtml−if expr="REQUEST.has_key('animalName')">
  <dtml−call expr="manage_changeProperties(animalName=REQUEST['animalName'])">
  <h1>The property 'animalName' has changed</h1>
<dtml−else>
  <h1>No properties were changed</h1>
</dtml−if>
```

In this example, the page will change a property depending on whether a certain name exists. The result of the *manage_changeProperties* method is not important and does not need to be shown to the user.

Another common usage of the *call* tag is calling methods that affect client behavior, like the `RESPONSE.redirect` method. In this example, you make the client redirect to a different page, to change the page that gets redirected, change the value for the "target" variable defined in the *let* tag:

```
<dtml−var standard_html_header>

<dtml−let target="'http://example.com/new_location.html'">
```

```
  <h1>This page has moved, you will now be redirected to the
  correct location.  If your browser does not redirect, click <a
  href="<dtml-var target>"><dtml-var target></a>.</h1>

  <dtml-call expr="RESPONSE.redirect(target)">

</dtml-let>

<dtml-var standard_html_footer>
```

In short, the *call* tag works exactly like the *var* tag with the exception that it doesn't insert the results of calling the variable.

The *Comment* Tag

DTML can be documented with comments using the *comment* tag:

```
<dtml-var standard_html_header>

<dtml-comment>

  This is a DTML comment and will be removed from the DTML code
  before it is returned to the client.  This is useful for
  documenting DTML code.  Unlike HTML comments, DTML comments
  are NEVER sent to the client.

</dtml-comment>

<!--

  This is an HTML comment, this is NOT DTML and will be treated
  as HTML and like any other HTML code will get sent to the
  client.  Although it is customary for an HTML browser to hide
  these comments from the end user, they still get sent to the
  client and can be easily seen by 'Viewing the Source' of a
  document.

-->

<dtml-var standard_html_footer>
```

The *comment* block is removed from DTML output.

In addition to documenting DTML you can use the *comment* tag to temporarily comment out other DTML tags. Later you can remove the *comment* tags to re−enable the DTML.

The *Tree* Tag

The *tree* tag lets you easily build dynamic trees in HTML to display hierarchical data. A *tree* is a graphical representation of data that starts with a "root" object that has objects underneath it often referred to as "branches". Branches can have their own branches, just like a real tree. This concept should be familiar to anyone who has used a file manager program like Microsoft Windows Explorer to navigate a file system. And, in fact, the left hand "navigation" view of the Zope management interface is created using the tree tag.

For example here's a tree that represents a collection of folders and sub−folders.

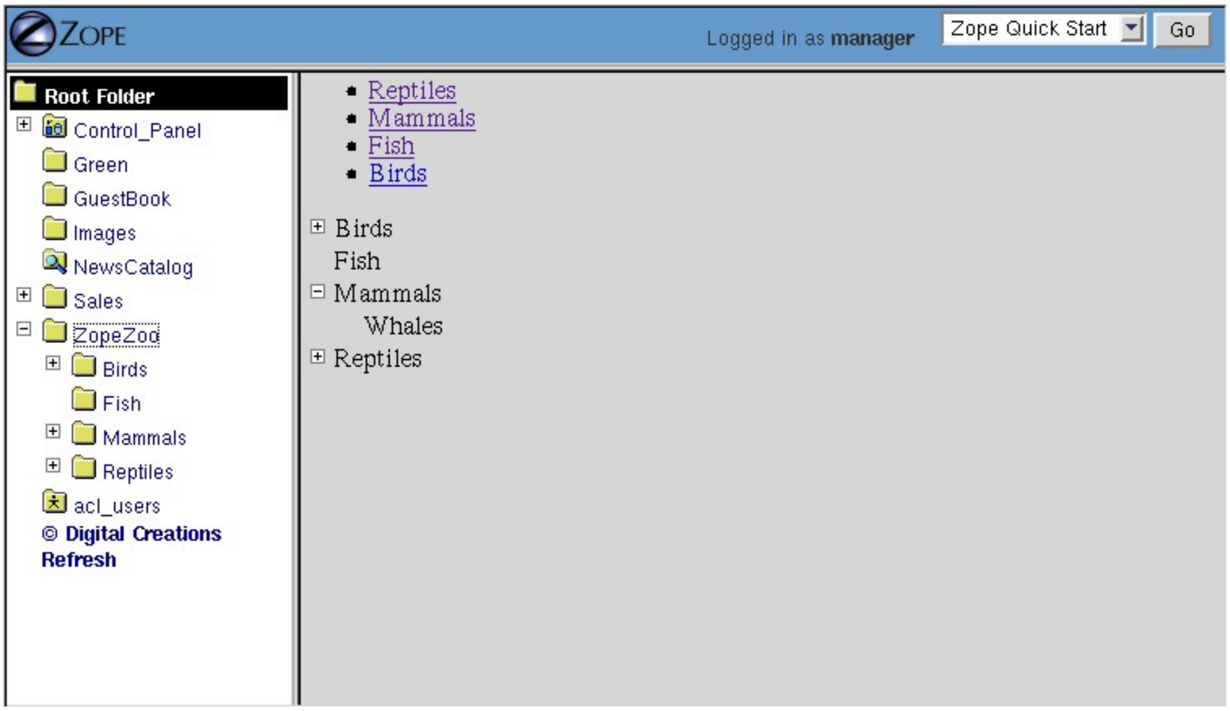


Figure 7−5 HTML tree generated by the tree tag.

Here's the DTML that generated this tree display:

```
<dtml-var standard_html_header>

<dtml-tree>

  <dtml-var getId>

</dtml-tree>

<dtml-var standard_html_footer>
```

The *tree* tag queries objects to find their sub−objects and takes care of displaying the results as a tree. The *tree* tag block works as a template to display nodes of the tree.

Now, since the basic protocol of the web, HTTP, is stateless, you need to somehow remember what state the tree is in every time you look at a page. To do this, Zope stores the state of the tree in a *cookie*. Because this tree state is stored in a cookie, only one tree can appear on a web page at a time, otherwise they will confusingly use the same cookie.

You can tailor the behavior of the *tree* tag quite a bit with *tree* tag attributes and special variables. Here is a sampling of *tree* tag attributes.

branches
: The name of the method used to find sub−objects. This defaults to *tpValues*, which is a method defined by a number of standard Zope objects.

leaves
: The name of a method used to display objects that do not have sub−object branches.

nowrap
: Either 0 or 1. If 0, then branch text will wrap to fit in available space, otherwise, text may be truncated. The default value is 0.

sort
: Sort branches before text insertion is performed. The attribute value is the name of the attribute that items should be sorted on.

assume_children
: Either 0 or 1. If 1, then all objects are assumed to have sub−objects, and will therefore always have a plus sign in front of them when they are collapsed. Only when an item is expanded will sub−objects be looked for. This could be a good option when the retrieval of sub−objects is a costly process. The defalt value is 0.

single
: Either 0 or 1. If 1, then only one branch of the tree can be expanded. Any expanded branches will collapse when a new branch is expanded. The default value is 0.

skip_unauthorized
: Either 0 or 1. If 1, then no errors will be raised trying to display sub−objects for which the user does not have sufficient access. The protected sub−objects are not displayed. The default value is 0.

Suppose you want to use the *tree* tag to create a dynamic site map. You don't want every page to show up in the site map. Let's say that you put a property on folders and documents that you want to show up in the site map.

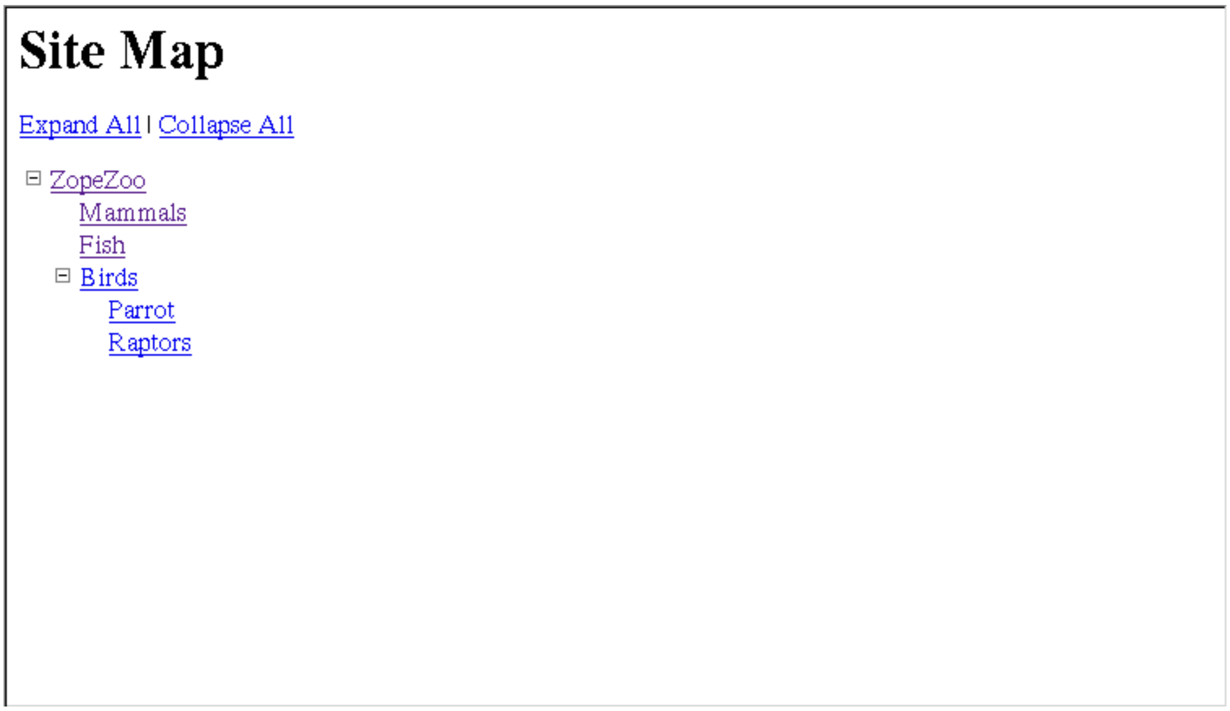
Let's first define a Script with the id of *publicObjects* that returns public objects:

```
## Script (Python) "publicObjects"
##
"""
Returns sub−folders and DTML documents that have a
true 'siteMap' property.
"""
results=[]
for object in context.objectValues(['Folder', 'DTML Document']):
    if object.hasProperty('siteMap') and object.siteMap:
        results.append(object)
return results
```

Now we can create a DTML Method that uses the *tree* tag and our Scripts to draw a site map:

```
<dtml−var standard_html_header>

<h1>Site Map</h1>

<p><a href="&dtml−URL0;?expand_all=1">Expand All</a> |
   <a href="&dtml−URL0;?collapse_all=1">Collapse All</a>
</p>

<dtml−tree branches="publicObjects" skip_unauthorized="1">
  <a href="&dtml−absolute_url;"><dtml−var title_or_id></a>
</dtml−tree>

<dtml−var standard_html_footer>
```

This DTML Method draws a link to all public resources and displays them in a tree. Here's what the resulting site map looks like.



Site Map

Expand All | Collapse All

⊟ ZopeZoo
    Mammals
    Fish
  ⊟ Birds
      Parrot
      Raptors

Figure 7−6 Dynamic site map using the tree tag.

For a summary of the *tree* tag arguments and special variables see Appendix A.

The *Return* Tag

In general DTML creates textual output. You can however, make DTML return other values besides text. Using the *return* tag you can make a DTML Method return an arbitrary value just like a Python or Perl−based Script.

Here's an example:

```
<p>This text is ignored.</p>

<dtml−return expr="42">
```

This DTML Method returns the number 42.

Another upshot of using the *return* tag is that DTML execution will stop after the *return* tag.

If you find yourself using the *return* tag, you almost certainly should be using a Script instead. The *return* tag was developed before Scripts, and is largely useless now that you can easily write scripts in Python and Perl.

The *Sendmail* Tag

The *sendmail* tag formats and sends a mail messages. You can use the *sendmail* tag to connect to an existing Mail Host, or you can manually specify your SMTP host.

Here's an example of how to send an email message with the *sendmail* tag:

```
<dtml−sendmail>
```

```
To: <dtml-var recipient>
Subject: Make Money Fast!!!!

Take advantage of our exciting offer now! Using our exclusive method
you can build unimaginable wealth very quickly. Act now!
</dtml-sendmail>
```

Notice that there is an extra blank line separating the mail headers from the body of the message.

A common use of the *sendmail* tag is to send an email message generated by a feedback form. The *sendmail* tag can contain any DTML tags you wish, so it's easy to tailor your message with form data.

The *Mime* Tag

The *mime* tag allows you to format data using MIME (Multipurpose Internet Mail Extensions). MIME is an Internet standard for encoding data in email message. Using the *mime* tag you can use Zope to send emails with attachments.

Suppose you'd like to upload your resume to Zope and then have Zope email this file to a list of potential employers.

Here's the upload form:

```
<dtml-var standard_html_header>

<p>Send you resume to potential employers</p>

<form method=post action="sendresume" ENCTYPE="multipart/form-data">
<p>Resume file: <input type="file" name="resume_file"></p>
<p>Send to:</p>
<p>
<input type="checkbox" name="send_to:list" value="jobs@yahoo.com">
  Yahoo<br>

<input type="checkbox" name="send_to:list" value="jobs@microsoft.com">
  Microsoft<br>

<input type="checkbox" name="send_to:list" value="jobs@mcdonalds.com">
  McDonalds</p>

<input type=submit value="Send Resume">
</form>

<dtml-var standard_html_footer>
```

Create another DTML Method called *sendresume* to process the form and send the resume file:

```
<dtml-var standard_html_header>

<dtml-if send_to>

  <dtml-in send_to>

    <dtml-sendmail smtphost="my.mailserver.com">
    To: <dtml-var sequence-item>
    Subject: Resume
    <dtml-mime type=text/plain encode=7bit>

    Hi, please take a look at my resume.
```

```
          <dtml-boundary type=application/octet-stream disposition=attachment
          encode=base64><dtml-var expr="resume_file.read()"></dtml-mime>
          </dtml-sendmail>

        </dtml-in>

        <p>Your resume was sent.</p>

      <dtml-else>

        <p>You didn't select any recipients.</p>

      </dtml-if>

      <dtml-var standard_html_footer>
```

This method iterates over the *sendto* variable and sends one email for each item.

Notice that there is no blank line between the `To:` header and the starting *mime* tag. If a blank line is inserted between them then the message will not be interpreted as a *multipart* message by the receiving mail reader.

Also notice that there is no newline between the *boundary* tag and the *var* tag, or the end of the *var* tag and the closing *mime* tag. This is important, if you break the tags up with newlines then they will be encoded and included in the MIME part, which is probably not what you're after.

As per the MIME spec, *mime* tags may be nested within *mime* tags arbitrarily.

The *Unless* Tag

The *unless* tag executes a block of code unless the given condition is true. The *unless* tag is the opposite of the *if* tag. The DTML code:

```
<dtml-if expr="not butter">
  I can't believe it's not butter.
</dtml-if>
```

is equivalent to:

```
<dtml-unless expr="butter">
  I can't believe it's not butter.
</dtml-unless>
```

What is the purpose of the *unless* tag? It is simply a convenience tag. The *unless* tag is more limited than the *if* tag, since it cannot contain an *else* or *elif* tag.

Like the *if* tag, calling the *unless* tag by name does existence checking, so:

```
<dtml-unless the_easter_bunny>
  The Easter Bunny does not exist or is not true.
</dtml-unless>
```

Checks for the existence of *the_easter_bunny* as well as its truth. While this example only checks for the truth of *the_easter_bunny*:

```
<dtml-unless expr="the_easter_bunny">
  The Easter Bunny is not true.
```

```
      </dtml-unless>
```

This example will raise an exception if *the_easter_bunny* does not exist.

Anything that can be done by the *unless* tag can be done by the *if* tag. Thus, its use is totally optional and a matter of style.

Batch Processing With The *In* Tag

Often you want to present a large list of information but only show it to the user one screen at a time. For example, if a user queried your database and got 120 results, you will probably only want to show them to the user a small batch, say 10 or 20 results per page. Breaking up large lists into parts is called *batching*. Batching has a number of benefits.

- The user only needs to download a reasonably sized document rather than a potentially huge document. This makes pages load faster since they are smaller.
- Because smaller batches of results are being used, often less memory is consumed by Zope.
- *Next* and *Previous* navigation interfaces makes scanning large batches relatively easy.

The *in* tag provides several variables to facilitate batch processing. Let's look at a complete example that shows how to display 100 items in batches of 10 at a time:

```
<dtml-var standard_html_header>

  <dtml-in expr="_.range(100)" size=10 start=query_start>

    <dtml-if sequence-start>

      <dtml-if previous-sequence>
        <a href="<dtml-var URL><dtml-var sequence-query
           >query_start=<dtml-var previous-sequence-start-number>">
           (Previous <dtml-var previous-sequence-size> results)
        </a>
      </dtml-if>

      <h1>These words are displayed at the top of a batch:</h1>
      <ul>

    </dtml-if>

      <li>Iteration number: <dtml-var sequence-item></li>

    <dtml-if sequence-end>

      </ul>
      <h4>These words are displayed at the bottom of a batch.</h4>

      <dtml-if next-sequence>
         <a href="<dtml-var URL><dtml-var sequence-query
            >query_start=<dtml-var
            next-sequence-start-number>">
         (Next <dtml-var next-sequence-size> results)
         </a>

      </dtml-if>

    </dtml-if>

  </dtml-in>
```

```
<dtml-var standard_html_footer>
```

Let's take a look at the DTML to get an idea of what's going on. First we have an *in* tag that iterates over 100 numbers that are generated by the *range* utility function. The *size* attribute tells the *in* tag to display only 10 items at a time. The *start* attribute tells the *in* tag which item number to display first.

Inside the *in* tag there are two main *if* tags. The first one tests special variable `sequence-start`. This variable is only true on the first pass through the in block. So the contents of this if tag will only be executed once at the beginning of the loop. The second *if* tag tests for the special variable `sequence-end`. This variable is only true on the last pass through the *in* tag. So the second *if* block will only be executed once at the end. The paragraph between the *if* tags is executed each time through the loop.

Inside each *if* tag there is another *if* tag that check for the special variables `previous-sequence` and `next-sequence`. The variables are true when the current batch has previous or further batches respectively. In other words `previous-sequence` is true for all batches except the first, and `next-sequence` is true for all batches except the last. So the DTML tests to see if there are additional batches available, and if so it draws navigation links.

The batch navigation consists of links back to the document with a *query_start* variable set which indicates where the *in* tag should start when displaying the batch. To better get a feel for how this works, click the previous and next links a few times and watch how the URLs for the navigation links change.

Finally some statistics about the previous and next batches are displayed using the `next-sequence-size` and `previous-sequence-size` special variables. All of this ends up generating the following HTML code:

```
<html><head><title>Zope</title></head><body bgcolor="#FFFFFF">

  <h1>These words are displayed at the top of a batch:</h1>
  <ul>
    <li>Iteration number: 0</li>
    <li>Iteration number: 1</li>
    <li>Iteration number: 2</li>
    <li>Iteration number: 3</li>
    <li>Iteration number: 4</li>
    <li>Iteration number: 5</li>
    <li>Iteration number: 6</li>
    <li>Iteration number: 7</li>
    <li>Iteration number: 8</li>
    <li>Iteration number: 9</li>
  </ul>
  <h4>These words are displayed at the bottom of a batch.</h4>

     <a href="http://pdx:8090/batch?query_start=11">
       (Next 10 results)
     </a>

</body></html>
```

Batch processing can be complex. A good way to work with batches is to use the Searchable Interface object to create a batching search report for you. You can then modify the DTML to fit your needs. This is explained more in Chapter 11, "Searching and Categorizing Content".

Exception Handling Tags

Zope has extensive exception handling facilities. You can get access to these facilities with the *raise* and *try* tags. For more information on exceptions and how they are raised and handled see a book on Python or you can read the online Python Tutorial.

The *Raise* Tag

You can raise exceptions with the *raise* tag. One reason to raise exceptions is to signal an error. For example you could check for a problem with the *if* tag, and in case there was something wrong you could report the error with the *raise* tag.

The *raise* tag has a type attribute for specifying an error type. The error type is a short descriptive name for the error. In addition, there are some standard error types, like *Unauthorized* and *Redirect* that are returned as HTTP errors. *Unauthorized* errors cause a log−in prompt to be displayed on the user's browser. You can raise HTTP errors to make Zope send an HTTP error. For example:

```
<dtml-raise type="404">Not Found</dtml-raise>
```

This raises an HTTP 404 (Not Found) error. Zope responds by sending the HTTP 404 error back to the client's browser.

The *raise* tag is a block tag. The block enclosed by the *raise* tag is rendered to create an error message. If the rendered text contains any HTML markup, then Zope will display the text as an error message on the browser, otherwise a generic error message is displayed.

Here is a *raise* tag example:

```
<dtml-if expr="balance >= debit_amount">

  <dtml-call expr="debitAccount(account, debit_amount)">

  <p><dtml-var debit_amount> has been deducted from your
  account <dtml-var account>.</p>

<dtml-else>

  <dtml-raise type="Insufficient funds">

    <p>There is not enough money in account <dtml-account>
    to cover the requested debit amount.</p>

  </dtml-raise>

</dtml-if>
```

There is an important side effect to raising an exception, exceptions cause the current transaction to be rolled back. This means any changes made by a web request to be ignored. So in addition to reporting errors, exceptions allow you to back out changes if a problem crops up.

The *Try* Tag

If an exception is raised either manually with the *raise* tag, or as the result of some error that Zope encounters, you can catch it with the *try* tag.

Exceptions are unexpected errors that Zope encounters during the execution of a DTML document or method. Once an exception is detected, the normal execution of the DTML stops. Consider the following example:

```
Cost per unit: <dtml-var
                     expr="_.float(total_cost/total_units)"
                     fmt=dollars-and-cents>
```

This DTML works fine if *total_units* is not zero. However, if *total_units* is zero, a *ZeroDivisionError* exception is raised indicating an illegal operation. So rather than rendering the DTML, an error message will be returned.

You can use the *try* tag to handle these kind of problems. With the *try* tag you can anticipate and handle errors yourself, rather than getting a Zope error message whenever an exception occurs.

The *try* tag has two functions. First, if an exception is raised, the *try* tag gains control of execution and handles the exception appropriately, and thus avoids returning a Zope error message. Second, the *try* tag allows the rendering of any subsequent DTML to continue.

Within the *try* tag are one or more *except* tags that identify and handle different exceptions. When an exception is raised, each *except* tag is checked in turn to see if it matches the exception's type. The first *except* tag to match handles the exception. If no exceptions are given in an *except* tag, then the *except* tag will match all exceptions.

Here's how to use the *try* tag to avoid errors that could occur in the last example:

```
<dtml-try>

  Cost per unit: <dtml-var
                       expr="_.float(total_cost/total_units)"
                       fmt="dollars-and-cents">

<dtml-except ZeroDivisionError>

  Cost per unit: N/A

</dtml-try>
```

If a *ZeroDivisionError* is raised, control goes to the *except* tag, and "Cost per unit: N/A" is rendered. Once the except tag block finishes, execution of DTML continues after the *try* block.

DTML's *except* tags work with Python's class−based exceptions. In addition to matching exceptions by name, the except tag will match any subclass of the named exception. For example, if *ArithmeticError* is named in a *except* tag, the tag can handle all *ArithmeticError* subclasses including, *ZeroDivisionError*. See a Python reference such as the online Python Library Reference for a list of Python exceptions and their subclasses. An *except* tag can catch multiple exceptions by listing them all in the same tag.

Inside the body of an *except* tag you can access information about the handled exception through several special variables.

error_type
: The type of the handled exception.

error_value
: The value of the handled exception.

error_tb
: The traceback of the handled exception.

You can use these variables to provide error messages to users or to take different actions such as sending email to the webmaster or logging errors depending on the type of error.

The *Try* Tag Optional *Else* Block

The *try* tag has an optional *else* block that is rendered if an exception didn't occur. Here's an example of how to use the *else* tag within the try tag:

```
<dtml-try>

  <dtml-call feedAlligators>

<dtml-except NotEnoughFood WrongKindOfFood>

  <p>Make sure you have enough alligator food first.</p>

<dtml-except NotHungry>

  <p>The alligators aren't hungry yet.</p>

<dtml-except>

  <p>There was some problem trying to feed the alligators.<p>
  <p>Error type: <dtml-var error_type></p>
  <p>Error value: <dtml-var error_value></p>

<dtml-else>

  <p>The alligator were successfully fed.</p>

</dtml-try>
```

The first *except* block to match the type of error raised is rendered. If an *except* block has no name, then it matches all raised errors. The optional *else* block is rendered when no exception occurs in the *try* block. Exceptions in the *else* block are not handled by the preceding *except* blocks.

The *Try* Tag Optional *Finally* Block

You can also use the *try* tag in a slightly different way. Instead of handling exceptions, the *try* tag can be used not to trap exceptions, but to clean up after them.

The *finally* tag inside the *try* tag specifies a cleanup block to be rendered even when an exception occurs.

The *finally* block is only useful if you need to clean up something that will not be cleaned up by the transaction abort code. The *finally* block will always be called, whether there is an exception or not and whether a *return* tag is used or not. If you use a *return* tag in the try block, any output of the *finally* block is discarded. Here's an example of how you might use the *finally* tag:

```
<dtml-call acquireLock>
<dtml-try>
    <dtml-call useLockedResource>
<dtml-finally>
    <!-- this always gets done even if an exception is raised -->
    <dtml-call releaseLock>
</dtml-try>
```

In this example you first acquire a lock on a resource, then try to perform some action on the locked resource. If an exception is raised, you don't handle it, but you make sure to release the lock before passing control off to an exception handler. If all goes well and no exception is raised, you still release the lock at the end of the *try* block by executing the *finally* block.

The *try/finally* form of the *try* tag is seldom used in Zope. This kind of complex programming control is often better done in Python or Perl.

Conclusion

DTML provides some very powerful functionality for designing web applications. In this chapter, we looked at the more advanced DTML tags and some of their options. A more complete reference can be found in Appendix A.

The next chapter teaches you how to become a Page Template wizard. While DTML is a powerful tool, Page Templates provide a more elegant solution to HTML generation.

Chapter 9. Advanced Page Templates

In Chapter 5, "Using Zope Page Templates" you learned the basics about Page Templates. In this chapter you'll learn about advanced techniques including new types of expressions and macros.

Advanced TAL

You've already learned about a few TAL statements. In this section we'll go over all TAL statements and their various options. Note, this material is covered more concisely in Appendix C, "Zope Page Templates Reference".

Advanced Content Insertion

You've already seen how `tal:content` and `tal:replace` work in Chapter 5, "Using Zope Page Templates". In this section you'll learn some advanced tricks for inserting content.

Inserting Structure

Normally, the `tal:replace` and `tal:content` statements escape HTML tags and entities in the text that they insert, converting `<` to `&lt;`, for instance. If you actually want to insert the unquoted text, you need to precede the expression with the `structure` keyword. For example:

```
<p replace="structure here/story">
  the <b>story</b>
</p>
```

This feature is useful when you are inserting a fragment of HTML or XML that is stored in a property or generated by another Zope object. For instance, you may have news items that contain simple HTML markup such as bold and italic text when they are rendered, and you want to preserve this when inserting them into a "Top News" page. In this case, you might write:

```
<p tal:repeat="newsItem here/topNews"
   tal:content="structure newsItem">
  A news item with<code>HTML</code> markup.
</p>
```

This will insert the news items including their HTML markup into paragraphs.

Dummy Elements

You can include page elements that are visible in the template but not in generated text by using the built−in variable `nothing`, like this:

```
<tr tal:replace="nothing">
  <td>10213</td><td>Example Item</td><td>$15.34</td>
</tr>
```

This can be useful for filling out parts of the page that will be populated with dynamic content. For instance, a table that usually has ten rows will only have one row in the template. By adding nine dummy rows, the template's layout will look more like the final result.

It's not always necessary to use the `tal:replace="nothing"` trick to get dummy content into your Page Template. For example, you've already seen that anything inside a `tal:content` or `tal:replace` element is normally removed when the template is rendered. In these cases you don't have do anything special to make sure that dummy content is removed.

Default Content

You can leave the contents of a tag along by using the `default` expression with `tal:content` or `tal:replace`. For example:

```
<p tal:content="default">Spam<p>
```

This renders to:

```
<p>Spam</p>
```

Most often you will want to selectively include default content, rather than always including it. For example:

```
<p tal:content="python:here.getFood() or default">Spam</p>
```

Note: Python expressions are explained later in the chapter. If the `getFood` method returns a true value than its result will be inserted into the paragraph, otherwise it's Spam for dinner.

Advanced Tag Repetition

You've already seen most of what you can do with the `tal:repeat` statement in Chapter 5, "Using Zope Page Templates". This section covers a few advanced features of the `tal:repeat` statement.

Repeat Variables

One topic that bear more explanation is repeat variables. Repeat variables provide information about the current repetition. These attributes are available on `repeat` variables:

- *index* − repetition number, starting from zero.
- *number* − repetition number, starting from one.
- *even* − true for even−indexed repetitions (0, 2, 4, ...).
- *odd* − true for odd−indexed repetitions (1, 3, 5, ...).
- *start* − true for the starting repetition (index 0).
- *end* − true for the ending, or final, repetition.
- *length* − length of the sequence, which will be the total number of repetitions.
- *letter* − count reps with lower−case letters: "a" − "z", "aa" − "az", "ba" − "bz", ..., "za" − "zz", "aaa" − "aaz", and so forth.
- *Letter* − upper−case version of *letter*.

You can access the contents of a repeat variable using path expressions or Python expressions. In path expressions, you write a three−part path consisting of the name `repeat`, the statement variable's name, and the name of the information you want, for example, `repeat/item/start`. In Python expressions, you use normal dictionary notation to get the repeat variable, then attribute access to get the information, for example, 'python:repeat['item'].start'.

Repetition Tips

Here are a couple practical tips that you may find useful. Some times you'd like to repeat a tag, but not have an enclosing tag. For example, you might want to repeat a number of paragraph tags, but there is no need to enclose them in another tag. You can do this with the `tal:omit−tag` statement:

```
<div tal:repeat="quote here/getQuotes"
     tal:omit-tag="">
  <p tal:content="quote">quotation</p>
</div>
```

The `tal:omit-tag` statement is described later in this chapter.

While it's been mentioned before, it's worth saying again: you can nest `tal:repeat` statements inside each other. Each `tal:repeat` statement must have a different repeat variable name. Here's an example that shows a math times−table:

```
<table border="1">
  <tr tal:repeat="x python:range(1, 13)">
    <div tal:repeat="y python:range(1, 13)"
         tal:omit-tag="">
      <td tal:content="python:'%d x %d = %d' % (x, y, x*y)">
        X x Y = Z
      </td>
    </div>
  </tr>
</table>
```

This example uses Python expressions, and the `tal:omit-tag` statement both of which are covered later in this chapter.

If you've done much work with the dtml−in DTML repetition statement, you will have encountered batching. Batching is the process of chopping up a large list into smaller lists. You typically use it to display a small number of items from a large list on a web page. Think of how a search engine batches its search results. The `tal:repeat` statement does not support batching, but Zope comes with a batching utility. See the section, "Batching" later in this chapter.

Another useful feature that isn't supplied by `tal:repeat` is sorting. If you want to sort a list you can either use write your own sorting script (which is quite easy in Python) or you can use the `sequence.sort` utility function. Here's an example of how to sort a list of objects by title, and then by modification date:

```
<table tal:define="objects here/objectValues;
                   sort_on python:(('title', 'nocase', 'asc'),
                                   ('bobobase_modification_time', 'cmp', 'desc'));
                   sorted_objects python:sequence.sort(objects, sort_on)">
  <tr tal:repeat="item sorted_objects">
    <td tal:content="item/title">title</td>
    <td tal:content="item/bobobase_modification_time">
      modification date</td>
  </tr>
</table>
```

This example tries to make things clearer by defining the sort arguments outside the `sort` function. You call the `sequence.sort` function takes a sequence and a description of how to sort it. In this example the description of how to sort the sequence is defined in the `sort_on` variable. See Appendix B, "API Reference" for more information on the powerful `sequence.sort` function.

Advanced Attribute Control

You've already met the `tal:attributes` statement. You can use it to dynamically replace tag attributes, for example, the `href` attribute on an `a` element. You can replace more than one attribute on a tag by separating attributes with semicolons:

```
<a href="link"
   tal:attributes="href here/getLink;
                   class here/getClass">link</a>
```

You can also define attributes with XML namespaces. For example:

```
<Description
    dc:Creator="creator name"
    tal:attributes="dc:Creator here/owner/getUserName">
  Description</Description>
```

Simply put the XML namespace prefix before the attribute name and you can create attributes with XML namespaces.

Defining Variables

You can define your own variable using the `tal:define` attribute. There are several reasons that you might want to do this. One reason is to avoid having to write long expressions repeatedly in a template. Another is to avoid having to call expensive methods repeatedly. You can define a variable once and then use it many times in a template. For example, here's a list that defines a variable and later tests it and repeats over it:

```
<ul tal:define="items container/objectIds"
    tal:condition="items">
  <li tal:repeat="item items">
    <p tal:content="item">id</p>
  </li>
</ul>
```

The `tal:define` statement creates the variable `items`, which you can use it anywhere in the `ul` tag. Notice also how you can have two TAL statement on the same `ul` tag. See the section, "Interactions Between TAL Statements" later in this chapter for more information about using more than one statement on a tag. In this case the first statement assigns the variable `items` and the second uses `items` in a condition to see whether or not it is false (in this case, an empty sequence) or true. If the `items` variable is false, then the `ul` tag is not shown.

Now, suppose that instead of simply removing the list when there are no items, you want to show a message. To do this, place the following before the list:

```
<h4 tal:condition="not:container/objectIds">There
Are No Items</h4>
```

The expression, `not:container/objectIds` is true when `container/objectIds` is false, and vice versa. See the section, "Not Expressions" later in this chapter for more information.

You can't use your `items` variable here, because it isn't defined yet. If you move the definition of `items` to the `h4` tag, then you can't use it in the `ul` tag any more, because it becomes a *local* variable of the `h4` tag. You could place the definition on some tag that enclosed both the `h4` and the `ul`, but there is a simpler solution. By placing the keyword `global` in front of the variable name, you can make the definition last from the `h4` tag to the bottom of the template:

```
<h4 tal:define="global items container/objectIds"
    tal:condition="not:items">There Are No Items</h4>
```

You can define more than one variable using `tal:define` by separating them with semicolons. For example:

```
<p tal:define="ids container/objectIds;
               title container/title">
```

You can define as many variables as you wish. Each variable can have its own global or local scope. You can also refer to earlier defined variables in later definitions. For example:

```
<p tal:define="title template/title;
               tlen python:len(title);">
```

With judicious use of `tal:define` you can improve the efficiency and readability of your templates.

Omitting Tags

You can remove tags with the `tal:omit−tag` statement. You will seldom need to use this TAL statement, but occasionally it's useful. The omit−tag attribute removes a tag, but does not affect the contents of the tag. For example:

```
<b tal:omit−tag=""><i>this</i> stays</b>
```

Renders to:

```
<i>this</i> stays
```

At this level of usage, `tal:omit−tag` operates almost like `tal:replace="default"`. However, `tal:omit−tag` is more useful when used in combination with other TAL statement such as `tal:repeat`. For example here's one way to create ten paragraph tags using `tal:repeat`:

```
<span tal:repeat="n python:range(10)"
      tal:omit−tag="">
  <p tal:content="n">1</p>
</span>
```

This will produce ten paragraph tags, however, the `span` tag will not appear in the output.

The `tal:omit−tag` attribute takes an expression, though normally you'll simply use an empty expression. If the expression is true or there is no expression then the statement tag is removed. If the expression is false, then the tag is not omitted. This allows you to selectively remove tags depending on dynamic circumstances.

Error Handling

If an error occurs in your page template, you can catch that error and show a useful error message to your user. For example, suppose your template defines a variable using form data:

```
...
<span tal:define="global prefs request/form/prefs"
      tal:omit−tag="" />
...
```

If Zope encounters a problem, like not being able to find the `prefs` variable in the form data, the entire page will break; you'll get an error page instead. Happily, you can avoid this kind of thing with limited error handling using the `tal:on−error` statement:

```
...
<span tal:define="global prefs here/scriptToGetPreferences"
      tal:omit−tag=""
      tal:on−error="string:An error occurred">
```

```
...
```

When an error is raised while rendering a template, Zope looks for a `tal:on-error` statement to handle the error. It first looks in the current tag, then on its enclosing tag, and so on until it reaches the top−level tag. When it finds an error handler, it replaces the contents of that tag with the error handling expression. In this case, the `span` tag will contain an error message.

Typically you'll define an error handler on a tag that encloses a logical page element, for example a table. If an error crops up drawing the table, then the error handler can simply omit the table from the page, or else replace it with an error message of some sort.

For more flexible error handling you can call a script. For example:

```
<div tal:on-error="structure here/handleError">
...
</div>
```

Any error that occurs inside the `div` will call the `handleError` script. Note that the `structure` option allows the script to return HTML. Your error handling script can examine the error and take various actions depending on the error. Your script gets access to the error through the `error` variable in the namespace. For example:

```
## Script (Python) "handleError"
##bind namespace=_
##
error=_['error']
if error.type==ZeroDivisionError:
    return "<p>Can't divide by zero.</p>"
else
    return """<p>An error occurred.</p>
              <p>Error type: %s</p>
              <p>Error value: %s</p>""" % (error.type,
                                           error.value)
```

Your error handling script can take all kinds of actions, for example, it might log the error by sending email.

The `tal:on-error` statement is not meant for general purpose exception handling. For example, you shouldn't validate form input with in. You should use a script for that, since scripts allow you to do powerful exception handling. The `tal:on-error` statement is for dealing with unusual problems that can occur when rendering templates.

Interactions Between TAL Statements

When there is only one TAL statement per element, the order in which they are executed is simple. Starting with the root element, each element's statements are executed, then each of its child elements is visited, in order, and their statements are executed, and so on.

However, it's possible to have more than one TAL statement on the same element. Any combination of statements may appear on the same element, except that the `tal:content` and `tal:replace` statements may not appear together.

When an element has multiple statements, they are executed in this order:

1. define
2. condition

3. repeat
4. content or replace
5. attributes
6. omit−tag

Since the `tal:on−error` statement is only invoked when an error occurs, it does not appear in the list.

The reasoning behind this ordering goes like this: you often want to set up variables for use in other statements, so define comes first. The very next thing to do is decide whether this element will be included at all, so condition is next; since the condition may depend on variables you just set, it comes after define. It is valuable to be able to replace various parts of an element with different values on each iteration of a repeat, so repeat comes before content, replace and attributes. Content and replace can't both be used on the same element so they occur at the same place. Omit−tag comes last since no other statement are likely to depend on it and since it should come after define and repeat.

Here's an example tag that includes several TAL statements:

```
<p tal:define="x /root/a/long/path/x | nothing"
   tal:condition="x"
   tal:content="x/txt"
   tal:attributes="class x/class">Ex Text</p>
```

Notice how the `tal:define` statement is executed first, and the other statements rely on its results.

There are three limits you should be aware of when combining TAL statements on elements:

1. Only one of each kind of statement can be used on a single tag. Since HTML does not allow multiple attributes with the same name. For example, you can't have two `tal:define` on the same tag.
2. Both of `tal:content` and `tal:replace` cannot be used on the same tag, since their functions conflict.
3. The order in which you write TAL attributes on a tag does not affect the order in which they execute. No matter how you arrange them, the TAL statements on a tag always execute in the fixed order described earlier.

If you want to override the ordering of TAL statements, you must do so by enclosing the element in another element, possibly `div` or `span`, and placing some of the statements on this new element. For example suppose you want to loop over a series of items but skip some. Here's an attempt to write a template that loops over the numbers zero to nine and skips three:

```
<!−− broken template −−>
<ul>
  <li tal:repeat="n python:range(10)"
      tal:condition="python:n != 3"
      tal:content="n">
    1
  </li>
</ul>
```

This template doesn't work because the the condition is tested before the repeat is executed. The upshot is that the `n` variable is not defined until after it is tested. Here's a way around this problem:

```
<ul>
  <div tal:repeat="n python:range(10)"
       tal:omit−tag="">
    <li tal:condition="python:n != 3"
        tal:content="n">
```

```
      1
    </li>
  </div>
</ul>
```

This template solves the problem by defining the `n` variable on an enclosing `div` tag. Notice that the `div` tag will not appear in the output on account of it's `tal:omit-tag` statement. This may be ugly, but it works. Perhaps future versions of Page Templates will solve this problem in a nicer fashion.

Form Processing

You can process forms in DTML using a common pattern called the "form/action pair". A form/action pair consists of two DTML methods or document: one that contains a form that collects input from the user, and one that contains an action that is taken on that input and returns the user a response. The form calls the action. See Chapter 4, "Dynamic Content with DTML" for more information on the form/action pattern.

Zope Page Templates don't work particularly well with the form/action pattern since it assumes that input processing and response presentation are handled by the same object (the action). Instead of the form/action pattern you should use form/action/response pattern with Page Templates. The form and response should be Page Templates and the action should be a script. The form template gathers the input and call the action script. The action script should process the input and return a response template. This pattern is more flexible than the form/action pattern since it allows the script to return any of a number of different response objects.

For example here's a part of a form template:

```
...
<form action="action">
  <input type="text" name="name">
  <input type="text" name="age:int">
  <input type="submit">
</form>
...
```

This form could be processed by this script:

```
## Script (Python) "action"
##parameters=name, age
##
container.addPerson(name, age)
return container.responseTemplate()
```

This script calls a method to process the inputs and then returns another template, the response. You can render a Page Template from Python by calling it. The response template typically contains an acknowledgment that the form has been correctly processed.

The action script can do all kinds of things. It can validate input, handle errors, send email, and more. Here's a sketch of how to validate input with an script:

```
## Script (Python) "action"
##
if not context.validateData(request):
    # if there's a problem return the form page template
    # along with an error message
    return context.formTemplate(error_message='Invalid data')

# otherwise return the thanks page
return context.responseTemplate()
```

This script validates the form input and returns the form template with an error message if there's a problem. You can pass Page Templates extra information with keyword arguments. The keyword arguments are available to the template as via the `options` built−in variable. So the form template in this example might include a section like this:

```
<b tal:condition="options/error_message | nothing"
   tal:content="options/error_message">
  Error message goes here.
</b>
```

This example shows how you can display an error message that is passed to the template via keyword arguments.

Depending on your application you may choose to redirect the user to a response Page Template instead of returning it directly. This results in twice as much network activity, but might be useful because it changes the URL displayed in the user's browser to the URL of the Page Template, rather than that of the action script.

If you insist on doing a crummy job of things, you can always create a lame version of the form−action pair using Page Templates. You should only do this when you don't care about error handling and when the response will always be the same, not matter what the user submits. Since Page Templates don't have an equivalent of dtml−call, you can use one of any number of hacks to call an input processing method without inserting its results. For example:

```
<span tal:define="unused here/processInputs"
      tal:omit-tag=""/>
```

This sample calls the `processInputs` method and assigns the result to the `unused` variable.

Expressions

You've already encountered Page Template expressions. Expressions provide values to template statements. For example, path expressions describe objects by giving them paths such as `request/form/age`, or `user/getUserName`. In this section you'll learn about all the different types of expressions, and variables.

Built−in Variables

Variables are names that you can use in expressions. You have already seen some examples of the built−in variables such as `template`, `user`, `repeat`, and `request`. Here is the complete list of the other built−in variables and their uses:

`nothing`
: A false value, similar to a blank string, that you can use in `tal:replace` or `tal:content` to erase a tag or its contents. If you set an attribute to `nothing`, the attribute is removed from the tag (or not inserted), unlike a blank string.

`default`
: A special value that doesn't change anything when used in `tal:replace`, `tal:content`, or `tal:attributes`. It leaves the template text in place.

`options`
: The keyword arguments, if any, that were passed to the template. Note: options are only available when a template is called from Python. When a template is rendered from the web, no options are present.

`attrs`

A dictionary of attributes of the current tag in the template. The keys are the attributes names, and the values are the original values of the attributes in the template. This variable is rarely needed.

`root`
: The root Zope object. Use this to get Zope objects from fixed locations, no matter where your template is placed or called.

`here`
: The object on which the template is being called. This is often the same as the *container*, but can be different if you are using acquisition. Use this to get Zope objects that you expect to find in different places depending on how the template is called. The `here` variable is analogous to the `context` variable in Python−based scripts.

`container`
: The container (usually a Folder) in which the template is kept. Use this to get Zope objects from locations relative to the template's permanent home. The `container` and `here` variables refer to the same object when a template is called from its normal location. However, when a template is applied to another object (for example, a ZSQL Method) the `container` and `here` will not refer to the same object.

`modules`
: The collection of Python modules available to templates. See the section on writing Python expressions.

You'll find examples of how to use these variables through out this chapter.

String Expressions

String expressions allow you to easily mix path expressions with text. All of the text after the leading `string:` is taken and searched for path expressions. Each path expression must be preceded by a dollar sign ('$'). Here are some examples:

```
"string:Just text. There's no path here."
"string:copyright $year, by me."
```

If the path expression has more than one part, or needs to be separated from the text that follows it, it must be surrounded by braces ('{}'). For example:

```
"string:Three ${vegetable}s, please."
"string:Your name is ${user/getUserName}!"
```

Notice how in the example above, you need to surround the `vegetable` path with braces so that Zope doesn't mistake it for `vegetables`.

Since the text is inside of an attribute value, you can only include a double quote by using the entity syntax `&quot;`. Since dollar signs are used to signal path expressions, a literal dollar sign must be written as two dollar signs ('$$'). For example:

```
"string:Please pay $$$dollars_owed"
"string:She said, &quot;Hello world.&quot;"
```

Some complex string formatting operations (such as search and replace or changing capitalization) can't easily be done with string expressions. For these cases, you should use Python expressions or Scripts.

Path Expressions

Path expressions refer to objects with a path that resembles a URL path. A path describes a traversal from object to object. All paths begin with a known object (such as a built−in variable, a repeat variable, or a user defined variable) and depart from there to the desired object. Here are some example paths expressions:

```
template/title
container/files/objectValues
user/getUserName
container/master.html/macros/header
request/form/address
root/standard_look_and_feel.html
```

With path expressions you can traverse from an object to its sub−objects including properties and methods. You can also use acquisition in path expressions. See the section entitled "Calling Scripts from the Web" in Chapter 8, "Advanced Zope Scripting" for more information on acquisition and path traversal.

Zope restricts object traversal in path expressions in the same way that it restricts object access via URLs. You must have adequate permissions to access an object in order to refer to it with a path expression. See Chapter 6, "Users and Security" for more information about object access controls.

Alternate Paths

The path `template/title` is guaranteed to exist every time the template is used, although it may be a blank string. Some paths, such as `request/form/x`, may not exist during some renderings of the template. This normally causes an error when Zope evaluates the path expression.

When a path doesn't exist, you may have a fall−back path or value that you would like to use instead. For instance, if `request/form/x` doesn't exist, you might want to use `here/x` instead. You can do this by listing the paths in order of preference, separated by vertical bar characters ('|'):

```
<h4 tal:content="request/form/x | here/x">Header</h4>
```

Two variables that are very useful as the last path in a list of alternates are `nothing` and `default`. For example, `default` tells `tal:content` to leave the dummy content. Different TAL statements interpret `default` and `nothing` differently. See Appendix C, "Zope Page Templates Reference" for more information.

You can also use a non−path expression as the final part in an alternate−path expression. For example:

```
<p tal:content="request/form/age|python:18">age</p>
```

In this example, if the `request/form/age` path doesn't exist, then the value is the number 18. This form allows you to specify default values to use which can't be expressed as paths. Note, you can only use a non−path expression as the last alternative.

You can also test the existence of a path directly with the *exists* expression type prefix. See the section "Exists Expressions" below for more information on exists expressions.

Not Expressions

Not expressions let you negate the value of other expressions. For example:

```
<p tal:condition="not:here/objectIds">
  There are no contained objects.
</p>
```

Not expressions return true when the expression they are applied to is false, and vice versa. In Zope, non−existent variables, zero, empty strings, empty sequences, nothing, and None are considered false, while everything else is true.

There isn't much reason to use not expressions with Python expressions since you can use the Python `not` keyword instead.

Nocall Expressions

An ordinary path expression tries to render the object that it fetches. This means that if the object is a function, Script, Method, or some other kind of executable thing, then expression will evaluate to the result of calling the object. This is usually what you want, but not always. For example, if you want to put a DTML Document into a variable so that you can refer to its properties, you can't use a normal path expression because it will render the Document into a string.

If you put the `nocall:` expression type prefix in front of a path, it prevents the rendering and simply gives you the object. For example:

```
<span tal:define="doc nocall:here/aDoc"
      tal:content="string:${doc/getId}: ${doc/title}">
Id: Title</span>
```

This expression type is also valuable when you want to define a variable to hold a function or class from a module, for use in a Python expression.

Nocall expressions can also be used on functions, rather than objects:

```
<p tal:define="join nocall:modules/string/join">
```

This expression defines the `join` variable as a function ('string.join'), rather than the result of calling a function.

Exists Expressions

An exists expression is true if its path exists, and otherwise is false. For example here's one way to display an error message only if it is passed in the request:

```
<h4 tal:define="err request/form/errmsg | nothing"
    tal:condition="err"
    tal:content="err">Error!</h4>
```

You can do the same thing more easily with an exists expression:

```
<h4 tal:condition="exists:request/form/errmsg"
    tal:content="request/form/errmsg">Error!</h4>
```

You can combine exists expressions with not expressions, for example:

```
<p tal:condition="not:exists:request/form/number">Please enter
a number between 0 and 5</p>
```

Note that in this example you can't use the expression, "not:request/form/number", since that expression will be true if the `number` variable exists and is zero.

Python Expressions

The Python programming language is a simple and expressive one. If you have never encountered it before, you should read one of the excellent tutorials or introductions available at the Python website.

A Page Template Python expression can contain anything that the Python language considers an expression. You can't use statements such as `if` and `while`. In addition, Zope imposes some security restrictions to keep you from accessing protected information, changing secured data, and creating problems such as infinite loops. See Chapter Chapter 8, "Advanced Zope Scripting" for more information on Python security restrictions.

Comparisons

One place where Python expressions are practically necessary is in `tal:condition` statements. You usually want to compare two strings or numbers, and there isn't any other way to do that without Python expressions. You can use the comparison operators `<` (less than), `>` (greater than), `==` (equal to), and `!=` (not equal to). You can also use the boolean operators `and`, `not`, and `or`. For example:

```
<p tal:repeat="widget widgets">
  <span tal:condition="python:widget.type == 'gear'>
  Gear #<span tal:replace="repeat/widget/number>1</span>:
  <span tal:replace="widget/name">Name</span>
  </span>
</p>
```

This example loops over a collection of objects, testing each object's `type` attribute.

Sometimes you want to choose different values inside a single statement based on one or more conditions. You can do this with the `test` function, like this:

```
You <span tal:define="name user/getUserName"
     tal:replace="python:test(name=='Anonymous User',
                              'need to log in', default)">
      are logged in as
      <span tal:replace="name">Name</span>
    </span>
```

The `test` function works like an if/then/else statement. See Appendix A, "DTML Reference" for more information on the `test` function. Here's another example of how you can use the `test` function:

```
<tr tal:define="oddrow repeat/item/odd"
    tal:attributes="class python:test(oddrow, 'oddclass',
                                      'evenclass')">
```

Without the `test` function you'd have to write two `tr` elements each with a different condition, one for even rows, and the other for odd rows.

Using other Expression Types

You can use other expression types inside of a Python expression. Each expression type has a corresponding function with the same name, including: `path()`, `string()`, `exists()`, and `nocall()`. This allows you to write expressions such as:

```
"python:path('here/%s/thing' % foldername)"
"python:path(string('here/$foldername/thing'))"
"python:path('request/form/x') or default"
```

The final example has a slightly different meaning than the path expression, "request/form/x | default", since it will use the default text if "request/form/x" doesn't exists *or* if it is false.

Getting at Zope Objects

Much of the power of Zope involves tying together specialized objects. Your Page Templates can use Scripts, SQL Methods, Catalogs, and custom content objects. In order to use these objects you have to know how to get access to them within Page Templates.

Object properties are usually attributes, so you can get a template's title with the expression "template.title". Most Zope objects support acquisition, which allows you to get attributes from "parent" objects. This means that the Python expression "here.Control_Panel" will acquire the Control Panel object from the root Folder. Object methods are attributes, as in "here.objectIds" and "request.set". Objects contained in a Folder can be accessed as attributes of the Folder, but since they often have Ids that are not valid Python identifiers, you can't use the normal notation. For example, you cannot use this Python expression:

```
"python:here.penguin.gif"'.
```

You must write:

```
"python:getattr(here, 'penguin.gif')"
```

since Python doesn't support attribute names with periods.

Some objects, such as `request`, `modules`, and Zope Folders support Python item access, for example:

```
request['URL']
modules['math']
here['thing']
```

When you use item access on a Folder, it doesn't try to acquire the name, so it will only succeed if there is actually an object with that Id contained in the Folder.

As shown in previous chapters, path expressions allow you to ignore details of how you get from one object to the next. Zope tries attribute access, then item access. You can write:

```
"here/images/penguin.gif"
```

instead of:

```
"python:getattr(here.images, 'penguin.gif')"
```

and:

```
"request/form/x"
```

instead of:

```
"python:request.form['x']"
```

The trade−off is that path expressions don't allow you to specify those details. For instance, if you have a form variable named "get", you must write:

```
"python:request.form['get']"
```

since this path expression:

```
"request/form/get"
```

will evaluate to the "get" *method* of the form dictionary.

If you prefer you can use path expressions inside Python expressions using the `path()` function, as described above.

Using Scripts

Script objects are often used to encapsulate business logic and complex data manipulation. Any time that you find yourself writing lots of TAL statements with complicated expressions in them, you should consider whether you could do the work better in a Script. If you have trouble understanding your template statements and expressions, then it's better to simplify your Page Template and use Scripts for the complex stuff.

Each Script has a list of parameters that it expects to be given when it is called. If this list is empty, then you can use the Script by writing a path expression. Otherwise, you will need to use a Python expression in order to supply the argument, like this:

```
"python:here.myscript(1, 2)"
"python:here.myscript('arg', foo=request.form['x'])"
```

If you want to return more than one item of data from a Script to a Page Template, it is a good idea to return it in a dictionary. That way, you can define a variable to hold all the data, and use path expressions to refer to each item. For example, suppose the `getPerson` script returns a dictionary with `name` and `age` keys:

```
<span tal:define="person here/getPerson"
      tal:replace="string:${person/name} is ${person/age}">
Name is 30</span> years old.
```

Of course, it's fine to return Zope objects and Python lists as well.

Calling DTML

Unlike Scripts, DTML Methods and Documents don't have an explicit parameter list. Instead, they expect to be passed a client, a mapping, and keyword arguments. They use these parameters to construct a namespace. See Chapter 7 for more information on explicitly calling DTML.

When the Zope publishes a DTML object through the web, it passes the context of the object as the client, and the REQUEST as the mapping. When one DTML object calls another, it passes its own namespace as the mapping, and no client.

If you use a path expression to render a DTML object, it will pass a namespace with `request`, `here`, and the template's variables already on it. This means that the DTML object will be able to use the same names as if it were being published in the same context as the template, plus the variable names defined in the template.

Python Modules

The Python language comes with a large number of modules, which provide a wide variety of capabilities to Python programs. Each module is a collection of Python functions, data, and classes related to a single purpose, such as mathematical calculations or regular expressions.

Several modules, including "math" and "string", are available in Python expressions by default. For example, you can get the value of pi from the math module by writing "python:math.pi". To access it from a path expression, however, you need to use the `modules` variable, "modules/math/pi".

The "string" module is hidden in Python expressions by the "string" expression type function, so you need to access it through the `modules` variable. You can do this directly in an expression in which you use it, or define a global variable for it, like this:

```
tal:define="global mstring modules/string"
tal:replace="python:mstring.join(slist, ':')"
```

In practice you'll rarely need to do this since you can use string methods most of the time rather than having to rely on functions in the string module.

Modules can be grouped into packages, which are simply a way of organizing and naming related modules. For instance, Zope's Python−based Scripts are provided by a collection of modules in the "PythonScripts" subpackage of the Zope "Products" package. In particular, the "standard" module in this package provides a number of useful formatting functions that are standard in the DTML "var" tag. The full name of this module is "Products.PythonScripts.standard", so you could get access to it using either of the following statements:

```
tal:define="global pps modules/Products/PythonScripts/standard"
tal:define="global pps python:modules['Products.PythonScripts.standard']"
```

Most Python modules cannot be accessed from Page Templates, DTML, or Scripts unless you add Zope security assertions to them. This procedure is outside the scope of this book. See the *Zope Developer's Guide* for more information.

Macros

So far, you've seen how page templates can be used to add dynamic behavior to individual web pages. Another feature of page templates is the ability to reuse look and feel elements across many pages.

For example, with Page Templates, you can have a site that has a standard look and feel. No matter what the "content" of a page, it will have a standard header, side−bar, footer, and/or other page elements. This is a very common requirement for web sites.

You can reuse presentation elements across pages with *macros*. Macros define a section of a page that can be reused in other pages. A macro can be an entire page, or just a chunk of a page such as a header or footer. After you define one or more macros in one Page Template, you can use them in other Page Templates.

Using Macros

You can define macros with tag attributes similar to TAL statements. Macro tag attributes are called Macro Expansion Tag Attribute Language (METAL) statements. Here's an example macro definition:

```
<p metal:define−macro="copyright">
  Copyright 2001, <em>Foo, Bar, and Associates</em> Inc.
</p>
```

This `metal:define−macro` statement defines a macro named "copyright". The macro consists of the `p` tag and its contents (including all contained tags).

Macros defined in a Page Template are stored in the template's `macro` attribute. You can use macros from other page template by referring to them through the `macros` attribute of the Page Template in which they are defined. For example, suppose the `copyright` macro is in a Page Template called "master_page". Here's how to use `copyright` macro from another Page Template:

```
<hr>
<b metal:use-macro="container/master_page/macros/copyright">
  Macro goes here
</b>
```

In this Page template, the `b` tag will be completely replaced by the macro when Zope renders the page:

```
<hr>
<p>
  Copyright 2001, <em>Foo, Bar, and Associates</em> Inc.
</p>
```

If you change the macro (for example, if the copyright holder changes their name) then all Page Templates that use the macro will automatically reflect the change.

Notice how the macro is identified by a path expression using the `metal:use-macro` statement. The `metal:use-macro` statement replaces the statement element with the named macro.

Macro Details

The `metal:define-macro` and `metal:use-macro` statements are pretty simple. However there are a few subtleties worth mentioning.

A macro's name must be unique within the Page Template in which it's defined. You can define more than one macro in a template, but they all need to have different names.

Normally you'll refer to a macro in a `metal:use-macro` statement with a path expression. However, you can use any expression type you wish so long as it returns a macro. For example:

```
<p metal:use-macro="python:here.getMacro()">
  Replaced with a dynamically determined macro,
  which is located by the getMacro script.
</p>
```

Using Python expressions to locate macros lets you dynamically vary which macro your template uses.

You can use the `default` variable with the `metal:use-macro` statement:

```
<p metal:use-macro="default">
  This content remains - no macro is used
</p>
```

The result is the same as using default with `tal:content` and `tal:replace`, the statement element doesn't change.

If you try to use the `nothing` variable with `metal:use-macro` you will get an error, since `nothing` is not a macro. If you want to use `nothing` to conditionally include a macro, you should instead enclose the `metal:use-macro` statement with a `tal:condition` statement.

Zope handles macros first when rendering your templates. Then Zope evaluates TAL expressions. For example, consider this macro:

```
<p metal:define-macro="title"
   tal:content="template/title">
  template's title
</p>
```

When you use this macro it will insert the title of the template in which the macro is used, *not* the title of the template in which the macro is defined. In other words, when you use a macro, it's like copying the text of a macro into your template and then rendering your template.

If you check the *Expand macros when editing* option on the Page Template *Edit* view, then any macros that you use will be expanded in your template's source. This is Zope's default behavior, and in general this is what you want, since it allows you to edit a complete and valid page. Sometimes, however, especially when you're editing in the ZMI, rather than using a WYSIWYG editing tool, it's more convenient not to expand macros when editing. In these cases, simply uncheck the option.

Using Slots

Macros are much more useful if you can override parts of them when you use them. You can do this by defining *slots* in the macro that you can fill in when you use the template. For example, consider a side bar macro:

```
<p metal:define-macro="sidebar">
  Links
  <ul>
    <li><a href="/">Home</a></li>
    <li><a href="/products">Products</a></li>
    <li><a href="/support">Support</a></li>
    <li><a href="/contact">Contact Us</a></li>
  </ul>
</p>
```

This macro is fine, but suppose you'd like to include some additional information in the sidebar on some pages. One way to accomplish this is with slots:

```
<p metal:define-macro="sidebar">
  Links
  <ul>
    <li><a href="/">Home</a></li>
    <li><a href="/products">Products</a></li>
    <li><a href="/support">Support</a></li>
    <li><a href="/contact">Contact Us</a></li>
  </ul>
  <span metal:define-slot="additional_info"></span>
</p>
```

When you use this macro you can choose to fill the slot like so:

```
<p metal:fill-slot="container/master.html/macros/sidebar">
  <b metal:fill-slot="additional_info">
    Make sure to check out our <a href="/specials">specials</a>.
  </b>
</p>
```

When you render this template the side bar will include the extra information that you provided in the slot:

```
<p>
  Links
  <ul>
```

```
      <li><a href="/">Home</a></li>
      <li><a href="/products">Products</a></li>
      <li><a href="/support">Support</a></li>
      <li><a href="/contact">Contact Us</a></li>
    </ul>
    <b>
      Make sure to check out our <a href="/specials">specials</a>.
    </b>
  </p>
```

Notice how the `span` element that defines the slot is replaced with the `b` element that fills the slot.

Customizing Default Presentation

A common use of slot is to provide default presentation which you can customize. In the slot example in the last section, the slot definition was just an empty `span` element. However, you can provide default presentation in a slot definition. For example, consider this revised sidebar macro:

```
<div metal:define-macro="sidebar">
  <p metal:define-slot="links">
  Links
  <ul>
    <li><a href="/">Home</a></li>
    <li><a href="/products">Products</a></li>
    <li><a href="/support">Support</a></li>
    <li><a href="/contact">Contact Us</a></li>
  </ul>
  </p>
  <span metal:define-slot="additional_info"></span>
</div>
```

Now the sidebar is fully customizable. You can fill the links slot to redefine the sidebar links. However, if you choose not to fill the `links` slot then you'll get the default links, which appear inside the slot.

You can even take this technique further by defining slots inside of slots. This allows you to override default presentation with a fine degree of precision. Here's a sidebar macro that defines slots within slots:

```
<div metal:define-macro="sidebar">
  <p metal:define-slot="links">
  Links
  <ul>
    <li><a href="/">Home</a></li>
    <li><a href="/products">Products</a></li>
    <li><a href="/support">Support</a></li>
    <li><a href="/contact">Contact Us</a></li>
    <span metal:define-slot="additional_links"></span>
  </ul>
  </p>
  <span metal:define-slot="additional_info"></span>
</div>
```

If you wish to customize the sidebar links you can either fill the `links` slot to completely override the links, or you can fill the `additional_links` slot to insert some extra links after the default links. You can nest slots as deeply as you wish.

Combining METAL and TAL

You can use both METAL and TAL statements on the same elements. For example:

```
<ul metal:define-macro="links"
    tal:repeat="link here/getLinks">
  <li>
    <a href="link url"
       tal:attributes="url link/url"
       tal:content="link/name">link name</a>
  </li>
</ul>
```

Since METAL statements are evaluated before TAL statements, there are no conflicts. This example is also interesting since it customizes a macro without using slots. The macro calls the `getLinks` Script to determine the links. You can thus customize your site's links be redefining the `getLinks` Script at different locations within your site.

It's not always easy to figure out the best way customize look and feel in different parts of your site. In general you should use slots to override presentation elements, and you should use Scripts to provide content dynamically. In the case of the links example, it's arguable whether links are content or presentation. Scripts probably provide a more flexible solution, especially if your site includes link content objects.

Whole Page Macros

Rather than using macros for chunks of presentation shared between pages, you can use macros to define entire pages. Slots make this possible. Here's an example macro that defines an entire page:

```
<html metal:define-macro="page">
  <head>
    <title tal:content="here/title">The title</title>
  </head>

  <body>
    <h1 metal:define-slot="headline"
        tal:content="here/title">title</h1>

    <p metal:define-slot="body">
      This is the body.
    </p>

    <span metal:define-slot="footer">
      <p>Copyright 2001 Fluffy Enterprises</p>
    </span>

  </body>
</html>
```

This macro defines a page with three slots, `headline`, `body`, and `footer`. Notice how the `headline` slot includes a TAL statement to dynamically determine the headline content.

You can then use this macro in templates for different types of content, or different parts of your site. For example here's how a template for news items might use this macro:

```
<html metal:use-macro="container/master.html/macros/page">

  <h1 metal:fill-slot="headline">
    Press Release:
    <span tal:replace="here/getHeadline">Headline</span>
  </h1>

  <p metal:fill-slot="body"
     tal:content="here/getBody">
```

```
      News item body goes here
    </p>

  </html>
```

This template redefines the `headline` slot to include the words, "Press Release" and call the `getHeadline` method on the current object. It also redefines the `body` slot to call the `getBody` method on the current object.

The powerful thing about this approach is that you can now change the `page` macro and the press release template will be automatically updated. For example you could put the body of the page in a table and add a sidebar on the left and the press release template would automatically use these new presentation elements.

This is a much more flexible solution to control page look and feel then the DTML `standard_html_header` and `standard_html_footer` solution. In fact, Zope comes with a stock page template in the root folder named `standard_template.pt` that includes a whole page macro with a `head` and `body` slot. Here's how you might use this macro in a template:

```
<html metal:use-macro="here/standard_template.pt/macros/page">
  <div metal:fill-slot="body">
    <h1 tal:content="here/title">Title</h1>
    <p tal:content="here/getBody">Body text goes here</p>
  </div>
</html>
```

Using the `standard_template.pt` macro is very similar to using other whole page macros. The only subtlety worth pointing out is the path used to locate the macro. In this example the path begins with `here`. This means that Zope will search for the `standard_template.pt` object using acquisition starting at the object that the template is applied to. This allows you to customize the look and feel of templates by creating custom `standard_template.pt` objects in various locations. This is exactly the same trick that you can use to customize look and feel by overriding `standard_html_header` and `standard_html_footer` in site locations. However, with `standard_template.pt` you have more choices. You can choose to start the path to the macro with `root` or with `container`, as well as with `here`. If the path begins with `root` then you will always get the standard template which is located in the root folder. If the path begins with `container` then Zope will search for a standard template using acquisition starting in the folder where the template is defined. This allows you to customize look and feel of templates, but does not allow you to customize the look and feel of different objects based on their location in the site.

Caching Templates

While rendering Page Templates normally is quite fast, sometimes it's not fast enough. For frequently accessed pages, or page that take a long time to render, you may want to trade some dynamic behavior for speed. Caching lets you do this. For more information on caching see the "Cache Manager" section of Chapter 3, "Basic Objects".

You can cache Page Templates using a cache manager in the same way that you cache other objects. To cache a Page Template, you must associate it with a cache manager. You can either do this by going to the *Cache* view of you Page Template and selecting the cache manager, or by going to the *Associate* view of your cache manager and locating your Page Template.

Here's an example of how to cache a Page Template. First create a Python−based script name `long.py` with these contents:

```
## Script (Python) "long.py"
```

```
##
for i in range(500):
  for j in range(500):
    for k in range(5):
      pass
return 'Done'
```

The purpose of this script is to take up a noticeable amount of execution time. Now create a Page Template that uses this script, for example:

```
<html>
  <body>
    <p tal:content="here/long.py">results</p>
  </body>
</html>
```

Now view this page. Notice how it takes a while to render. Now let's radically improve its rendering time with caching. Create a Ram Cache Manager if you don't already have one. Make sure to create it within the same folder as your Page Template, or in a higher level. Now visit the *Cache* view of your Page Template. Choose the Ram Cache Manager you just created and click *Save Changes*. Click the *Cache Settings* link to see how your Ram Cache Manager is configured. By default, your cache stores objects for one hour (3600 seconds). You may want to adjust this number depending on your application. Now return to your Page Template and view it again. It should take a while for it to render. Now reload the page, and watch it render immediately. You can reload the page again and again, and it will always render immediately since the page is now cached.

If you change your Page Template, then it will be removed from the cache. So the next time you view it, it will take a while to render. But after that it will render quickly since it will be cached again.

Caching is a simple but very powerful technique for improving performance. You don't have to be a wizard to use caching, and it can provide great speed−ups. It's well worth your time to use caching for performance−critical applications.

Page Template Utilities

Zope Page Templates are powerful but simple. Unlike DTML, Page Templates don't give you a lot of convenience features for things like batching, drawing trees, sorting, etc. The creators of Page Templates wanted to keep them simple. However, you may miss some of the built−in features that DTML provides. To address these needs, Zope comes with utilities designed to enhance Page Templates.

Batching Large Sets of Information

When a user queries a database and gets a hundred results, it's often better to show them several pages with only twenty results per page, rather than putting all the results on one page. Breaking up large lists into smaller lists is called *batching*.

Unlike DTML, which provides batching built into the language, Page Templates support batching by using a special `Batch` object that comes from the `ZTUtils` utility module. See Appendix B, "API Reference", for more information on the `ZTUtils` Python module.

Here's a simple example, showing how to create a `Batch` object:

```
  <ul tal:define="lots python:range(100);
                  batch python:modules['ZTUtils'].Batch(lots,
                                                         size=10,
                                                         start=0)">
```

```
    <li tal:repeat="num batch"
        tal:content="num">0
    </li>
  </ul>
```

This example renders a list with 10 items (in this case, the numbers 0 through 9). The `Batch` object chops a long list up into groups or batches. In this case it broke a one hundred item list up into batches of ten items.

You can display a different batch of ten items by passing a different start number:

```
<ul tal:define="lots python:range(100);
                batch python:modules['ZTUtils'].Batch(lots,
                                                      size=10,
                                                      start=13)">
```

This batch starts with the fourteenth item and ends with the twenty third item. In other words, it displays the numbers 13 through 22. It's important to notice that the batch `start` argument is the *index* of the first item. Indexes count from zero, rather than from one. So index 13 points to the fourteenth item in the sequence. Python uses indexes to refer to list items.

Normally when you use batches you'll want to include navigation elements on the page to allow users to go from batch to batch. Here's a full−blow batching example that shows how to navigate between batches:

```
<html>
  <head>
    <title tal:content="template/title">The title</title>
  </head>
  <body tal:define="employees  here/getEmployees;
                    start python:path('request/start') or 0;
                    batch python:modules['ZTUtils'].Batch(employees,
                                                          size=10,
                                                          start=start);
                    previous python:batch.previous;
                    next python:batch.next">

  <p>
    <a tal:condition="previous"
       tal:attributes="href string:${request/URL0}?start:int=${previous/first}"
       href="previous_url">previous</a>
    <a tal:condition="next"
       tal:attributes="href string:${request/URL0}?start:int=${next/first}"
       href="next_url">next</a>
  </p>

  <ul tal:repeat="employee batch" >
    <li>
      <span tal:replace="employee/name">Bob Jones</span>
      makes $<span tal:replace="employee/salary">100,000</span>
      a year.
    </li>
  </ul>

  </body>
</html>
```

This example iterates over batches of results from the `getEmployees` ZSQL Method. It draws a *previous* and a *next* link as necessary to allow you to page through all the results a batch at a time.

Take a look at the `tal:define` statement on the `body` element. It defines a bunch of batching variables. The `employees` variable is a potentially big list of employee objects returned by the `getEmployees`

ZSQL Method. The second variable, `start`, is either set to the value of `request/start` or to zero if there is no `start` variable in the request. The `start` variable keeps track of where you are in the list of employees. The `batch` variable is a batch of ten items from the lists of employees. The batch starts at the location specified by the `start` variable. The `previous` and `next` variables refer to the previous and next batches (if any). Since all these variables are defined on the `body` element, they are available to all elements inside the body.

Next let's look at the navigation links. They create hyper links to browse previous and next batches. The `tal:condition` statement first tests to see if there is a previous and next batch. If there is a previous or next batch, then the link is rendered, otherwise there is not link. The `tal:attributes` statement creates a link to the previous and next batches. The link is simply the URL or the current page ('request/URL0') along with a query string indicating the start index of the batch. For example, if the current batch starts with index 10, then the previous batch will start with an index of 0. The `first` variable of a batch gives its staring index, so in this case, `previous.start` would be 0.

It's not important to fully understand the workings of this example. Simply copy it, or use a batching example created by the *Z Search Interface*. Later when you want to do more complex batching you can experiment by changing the example code. Don't forget to consult Appendix B, "API Reference" for more information on the `ZTUtils` module and `Batch` objects.

Miscellaneous Utilities

Zope provides a couple Python modules which may come in handy when using Page Templates. The `string`, `math`, and `random` modules can be used in Python expressions for string formatting, math function, and pseudo−random number generation. These same modules are available from DTML and Python−based scripts. See Appendix B, "API Reference" for more information on these modules.

The `Products.PythonScripts.standard` module is designed to provide utilities to Python−based scripts, but it's also useful for Page Templates. It includes various string and number formatting functions. See Appendix B, "API Reference" for more information.

As mentioned earlier in the chapter, the `sequence` module provides a handy `sort` function. See Appendix B, "API Reference" for the details.

Finally the `AccessControl` module includes a function and a class which you'll need if you want to test access and to get the authenticated user. See Appendix B, "API Reference" for more information.

Conclusion

This chapter covers all the nooks and crannies of Page Templates, and after reading it you may feel a little overwhelmed. Don't worry, you don't need to know everything in this chapter to effectively use Page Templates. You should understand the different path types and macros, but you can come back to the rest of the material when you need it. The advanced features that you've learned about in this chapter are there for you when you need them. It's encouraging to know that when you're ready you can do some pretty impressive tricks with Page Templates.

Chapter 10: Advanced Zope Scripting

Zope manages your presentation, logic and data with objects. So far, you've seen how Zope can manage presentation with DTML, and data with files and images. This chapter shows you how to add *Script* objects that allows you to write scripts in Python, and Perl through your web browser.

What is *logic* and how does it differ from presentation? Logic provides the actions that change objects, send messages, test conditions and respond to events, whereas presentation formats and displays information and reports. Typically you will use DTML to handle presentation, and Zope scripting with Python and Perl to handle logic.

Zope Scripts

Zope *Script* objects are objects that encapsulate a small chunk of code written in a programming language. Currently, Zope provides *Python−based Scripts*, which are written in the Python language, and *Perl−based Scripts* which are written in the Perl language. Script objects are new as of Zope 2.3, and are the preferred way to write programming logic in Zope.

So far in this book you have heavily used DTML Methods and Documents to create simple web applications in Zope. DTML allows you to perform simple scripting operations such as string manipulation. For the most part, however, DTML Methods should be used for presentation. DTML Methods are explained in Chapters 4, "Dynamic Content with DTML", and Chapter 8, "Variables and Advanced DTML".

Here is an overview of Zope's scripts:

Python−based Scripts
: You can use Python, a general purpose scripting language, to control Zope objects and perform other tasks. These Scripts give you general purpose programming facilities within Zope.

Perl−based Scripts
: You can use Perl, a powerful text processing language, to script Zope objects and access Perl libraries. These scripts offer benefits similar to those of Python−based Scripts, but may be more appealing for folks who know Perl but not Python, or who want to use Perl libraries for which there are no Python equivalents.

You can add these scripts to your Zope application just like any other object.

Calling Scripts

Zope scripts are called from the web or from other scripts or objects. Almost any type of script can be called by any other type of object; you can call a Python−based Script from a DTML Method, or a built−in method from a Perl−based Script. In fact scripts can call scripts which call other scripts, and so on. As you saw in Chapter 4, "Dynamic Content with DTML", you can replace a script with a script implemented in another language transparently. For example if you're using Perl to perform a task, but later decide that it would be better done in Python, you can usually replace the script with a Python−based Script with the same id.

When you call a script, the way that you call it gives the script a context in which to execute. A script's context is important. For example, when you call a script you usually want to single out some object that is central to the script's task. You would call the script in the context of the object on which you want it to carry out its task. It is simpler to just say that you are calling the script *on* the object.

Calling Scripts From the Web

You can call a script directly from with web by visiting its URL. You can call a single script on different objects by using different URLS. This works because by using different URLs you can give your scripts different contexts, and scripts can operate differently depending on their context. This is a powerful feature that enables you to apply logic to objects like documents or folders without having to embed the actual code within the object.

To call a script on an object from the web, simply visit the URL of the object, followed by the name of the script. This places the script in the context of your object. For example suppose you have a collection of objects and scripts as shown in Figure 8−1.

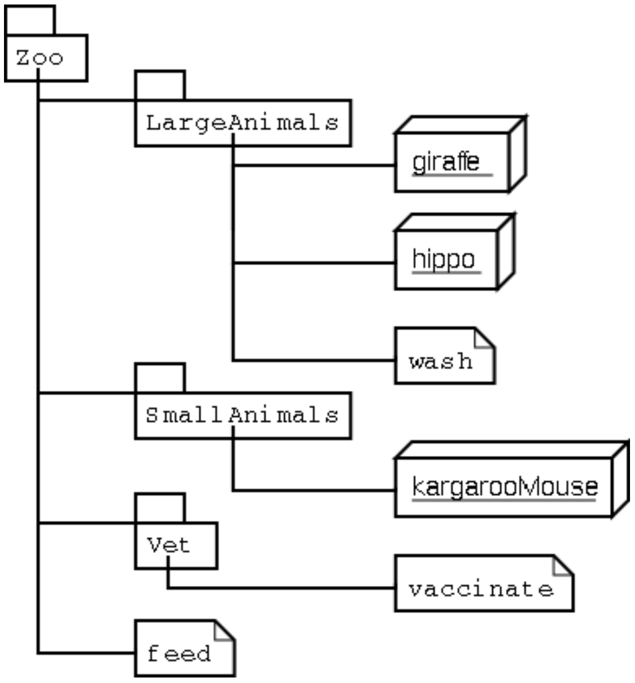


Figure 8−1 A collection of objects and scripts

To call the *feed* script on the *hippo* object you would visit the URL *Zoo/LargeAnimals/hippo/feed* To call the *feed* script on the *kangarooMouse* object you can visit the URL *Zoo/SmallAnimals/kangarooMouse/feed*. These URLs place the *feed* script in the context of the *hippo* and *kargarooMouse* objects, respectively.

Zope uses a URL as a map to find what object and what script you want to call.

Zope breaks apart the URL and compares it to the object hierarchy, working backwards until it finds a match for each part. This process is called *URL traversal*. For example, when you give Zope the URL *Zoo/LargeAnimals/hippo/feed*, it starts at the root folder and looks for an object named *Zoo*. It then moves to the *Zoo* folder and looks for an object named *LargeAnimals*. It moves to the *LargeAnimals* folder and looks for an object named *hippo*. It moves to the *hippo* object and looks for an object named *feed*. The *feed* script can't be found in the *hippo* object and is located in the *Zoo* folder by a process called *acquisition*.

Acquisition does two things. First it tries to find the object in the current object's containers. If that doesn't work it backs up along the URL path and tries again. In this example Zope first looks for the *feed* object in *hippo*, then it goes to the first container, *LargeAnimals*, and then to the next container, *Zoo*, where *feed* is finally found.

Now Zope has reached the end of the URL. It calls the last object found, *feed*. The *feed* script operates on its context which is the second to last object found, the *hippo* object. This is how the *feed* script is called on the *hippo* object.

Likewise you can call the *wash* method on the *hippo* with the URL *Zoo/LargeAnimals/hippo/wash*. In this case Zope acquires the *wash* method from the *LargeAnimals* folder.

More complex arrangements are possible. Suppose you want to call the *vaccinate* script on the *hippo* object. What URL can you use? If you visit the URL *Zoo/LargeAnimals/hippo/vaccinate* Zope will not be able to find the *vaccinate* script since it isn't in any of the *hippo* object's containers.

The solution is to give the path to the script as part of the URL. This way, when Zope uses acquisition to find the script it will find the right script as it backtracks along the URL. The URL to vaccinate the hippo is *Zoo/Vet/LargeAnimals/hippo/vaccinate*. Likewise, if you want to call the *vaccinate* script on the *kargarooMouse* object you should use the URL *Zoo/Vet/SmallAnimals/kargarooMouse/vaccinate*.

Let's follow along as Zope traverses the URL *Zoo/Vet/LargeAnimals/hippo/vaccinate*. Zope starts in the root folder and looks for an object named *Zoo*. It moves to the *Zoo* folder and looks for an object named *Vet*. It moves to the *Vet* folder and looks for an object named *LargeAnimals*. The *Vet* folder doesn't contain an object with that name, but it can acquire the *LargeAnimals* folder from its container, *Zoo* folder. So it moves to the *LargeAnimals* folder and looks for an object named *hippo*. It then moves to the *hippo* object and looks for an object named *vaccinate*. Since the *hippo* object does not contain a *vaccinate* object and neither do any of its containers, Zope backtracks along the URL path trying to find a *vaccinate* object. First it backs up to the *LargeAnimals* folder where *vaccinate* still can't be found. Then it backs up to the *Vet* folder. Here it finds a *vaccinate* script in the *Vet* folder. Since Zope has now come to the end of the URL, it calls the *vaccinate* script in the context of the *hippo* object.

When Zope looks for a sub−object during URL traversal, it first looks for the sub−object in the current object. If it can't find it in the current object it looks in the current object's containers. If it still can't find the sub−object, it backs up along the URL path and searches again. It continues this process until it either finds the object or raises an error if it can't be found.

This is a very useful mechanism, and it allows you to be quite expressive when you compose URLs. The path that you tell Zope to take on its way to an object will determine how it uses acquisition to look up the object's scripts.

Calling Scripts from other Objects

You can call scripts from other objects. For example, it is common to call scripts from DTML Methods.

As you saw in Chapter 8, "Variables and Advanced DTML", you can call Zope scripts from DTML with the *call* tag. For example:

```
<dtml−call updateInfo>
```

DTML will call the *updateInfo* script. You don't have to specify if the script is implemented in Perl, Python, or any other language (you can also call other DTML objects and SQL Methods this way).

If the *updateInfo* script requires parameters, you must either choose a name for the DTML namespace binding (see Binding Variables below) so that the parameters will be looked up in the namespace, or you must pass the parameters in an expression, like this:

```
<dtml−call expr="updateInfo(color='brown', pattern='spotted')">
```

Calling scripts from Python and Perl works the same way, except that you must always pass script parameters when you call a script from Python or Perl. For example here's how you might call the *updateInfo* script from Python:

```
context.updateInfo(color='brown',
                   pattern='spotted')
```

From Perl you could do the same thing using standard Perl semantics for calling scripts:

```
$self->updateInfo(color => 'brown',
                  pattern => 'spotted');
```

Each scripting language has a different way of writing a script call, but you don't have to know what language is used in the script you are calling. Effectively Zope objects can have scripts implemented in several different languages. But when you call a script you don't have to know how it's implemented, you just need to pass the appropriate parameters.

Zope locates the scripts you call using acquisition the same way it does when calling scripts from the web. Returning to our hippo feeding example of the last section, let's see how to vaccinate a hippo from Python and Perl. Figure 8−2 shows a slightly updated object hierarchy that contains two scripts, *vaccinateHippo.py* and *vaccinateHippo.pl*.

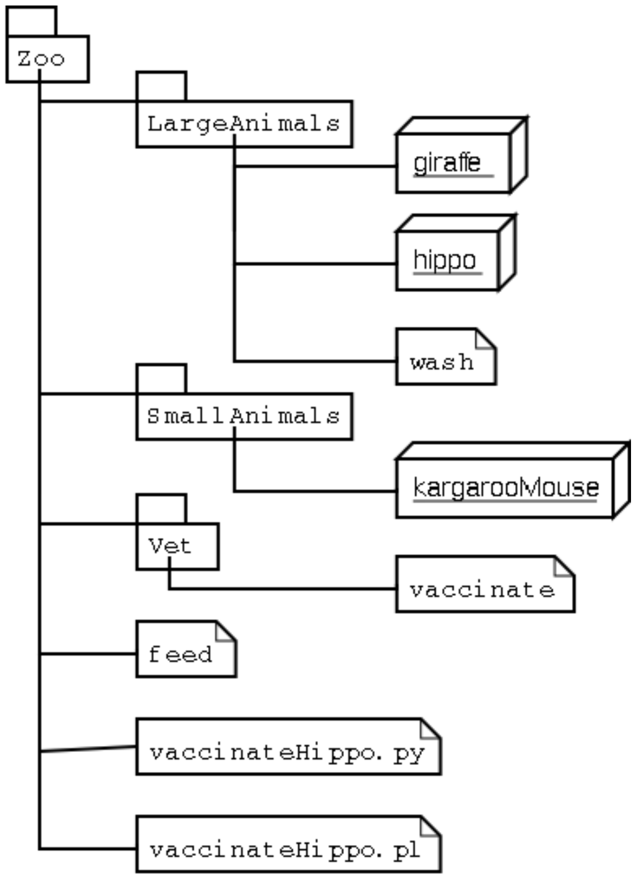


Figure 8−2 A collection of objects and scripts

Suppose *vaccinateHippo.py* is a Python script. Here's how you call the *vaccinate* script on the *hippo* object from Python:

```
context.Vet.LargeAnimals.hippo.vaccinate()
```

In other words you simply access the object using the same acquisition path as you would use if calling it from the web. Likewise in Perl you could say:

```
$self−>Vet−>LargeAnimals−>hippo−>vaccinate();
```

Using scripts from other scripts is very similar to calling scripts from the web. The semantics differ slightly but the same acquisition rules apply. Later on in this chapter, you'll see more examples of how scripts in both Perl and Python work.

Passing Parameters to Scripts

All scripts can be passed parameters. A parameter gives a script more information about what to do. When you call a script from the web, Zope will try to find the script's parameters in the web request and pass them to your script. For example if you have a script with parameters *dolphin* and *REQUEST* Zope will look for *dolphin* in the web request, and will pass the request itself as the *REQUEST* parameter. In practical terms this means that it is easy to do form processing in your script. For example here is a form:

```
<form action="actionScript">
Name <input type="text" name="name"><br>
Age <input type="text" name="age:int"><br>
<input type="submit">
</form>
```

You can easily process this form with a script named *actionScript* that includes *name* and *age* in its parameter list:

```
## Script (Python) "actionScript"
##parameters=name, age
##
"Process form"
context.processName(name)
context.processAge(age)
return context.responseMessage()
```

There's no need to process the form manually to extract values from it. Form elements are passed as strings, or lists of strings in the case of check boxes, and multiple−select input.

In addition to form variables, you can specify any request variables as script parameters. For example, to get access to the request and response objects just include `REQUEST` and `RESPONSE` in your list of parameters. Request variables are detailed more fully in Appendix B.

One thing to note is that the *context* variable refers to the object that your script is called on. This works similarly in Perl−based Scripts, for example:

```
my $self = shift;
$self−>processName($name);
$self−>processAge($age);
return $context−>responseMessage();
```

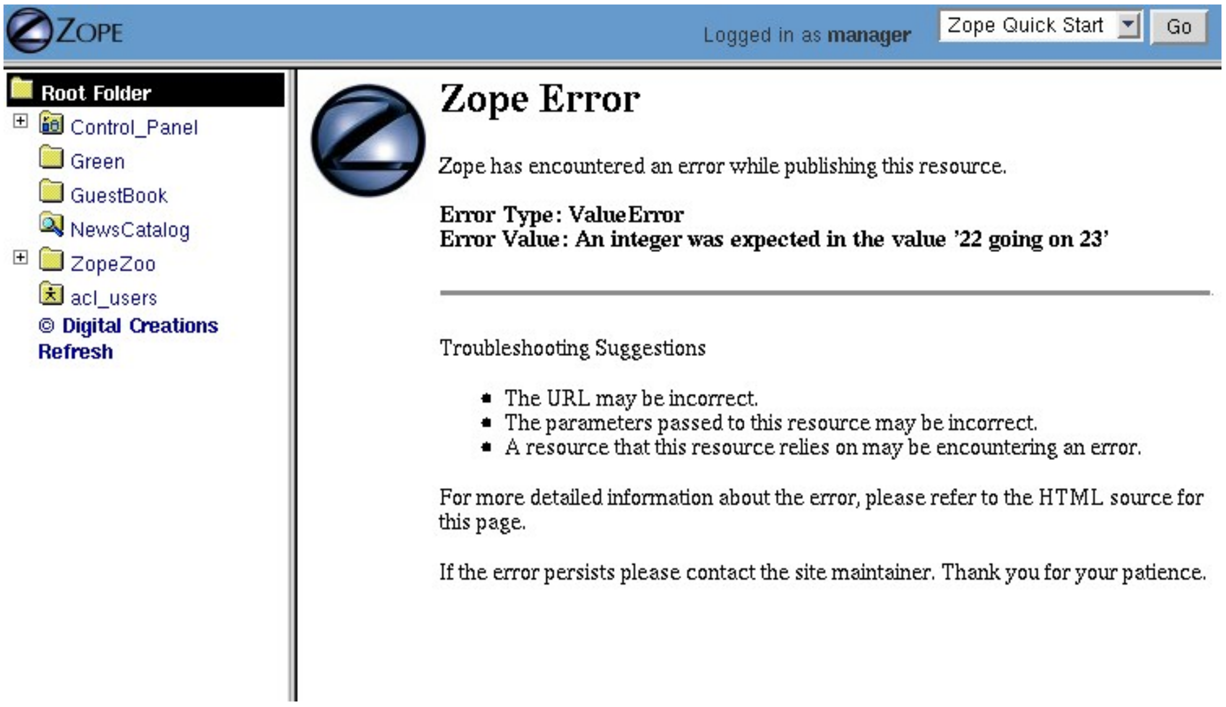
In the Python version of the example, there is a subtle problem. You are probably expecting an integer rather than a string for age. You could manually convert the string to an integer using the Python *int* built−in:

```
age=int(age) # covert a string to an integer
```

But this manual conversion may be inconvenient. Zope provides a way for you to specify form input types in the form, rather than in the processing script. Instead of converting the *age* variable to an integer in the processing script, you can indicate that it is an integer in the form:

```
Age <input type="text" name="age:int">
```

The `:int` appended to the form input name tells Zope to automatically convert the form input to an integer. If the user of your form types something that can't be converted to an integer (such as "22 going on 23") then Zope will raise an exception as shown in Figure 8−3.



Zope Error

Zope has encountered an error while publishing this resource.

Error Type: ValueError
Error Value: An integer was expected in the value '22 going on 23'

Troubleshooting Suggestions

- The URL may be incorrect.
- The parameters passed to this resource may be incorrect.
- A resource that this resource relies on may be encountering an error.

For more detailed information about the error, please refer to the HTML source for this page.

If the error persists please contact the site maintainer. Thank you for your patience.

Figure 8−3 Parameter conversion error

It's handy to have Zope catch conversion errors, but you may not like Zope's error messages. You should avoid using Zope's converters if you want to provide your own error messages.

Zope can perform many parameter conversions. Here is a list of Zope's basic parameter converters.

boolean
: Converts a variable to true or false. Variables that are 0, None, an empty string, or an empty sequence are false, all others are true.

int
: Converts a variable to an integer.

long
: Converts a variable to a long integer.

float
: Converts a variable to a floating point number.

string
: Converts a variable to a string. Most variables are strings already so this converter is seldom used.

text
: Converts a variable to a string with normalized line breaks. Different browsers on various platforms encode line endings differently, so this script makes sure the line endings are consistent, regardless of how they were encoded by the browser.

list
: Converts a variable to a Python list.

tuple
: Converts a variable to a Python tuple. A tuple is like a list, but cannot be modified.

tokens
: Converts a string to a list by breaking it on white spaces.

lines
: Converts a string to a list by breaking it on new lines.

date
: Converts a string to a *DateTime* object. The formats accepted are fairly flexible, for example `10/16/2000`, `12:01:13 pm`.

required
: Raises an exception if the variable is not present.

ignore_empty
: Excludes the variable from the request if the variable is an empty string.

These converters all work in more or less the same way to coerce a string form variable into a specific type. You may recognize these converters from Chapter 3, "Using Basic Zope Objects", where we discussed properties. These converters are used by Zope's property facility to convert properties to the right type.

The *list* and *tuple* converters can be used in combination with other converters. This allows you to apply additional converters to each element of the list or tuple. Consider this form:

```
<form action="processTimes">

<p>I would prefer not to be disturbed at the following
times:</p>

<input type="checkbox" name="disturb_times:list:date"
value="12:00 AM"> Midnight<br>

<input type="checkbox" name="disturb_times:list:date"
value="01:00 AM"> 1:00 AM<br>

<input type="checkbox" name="disturb_times:list:date"
value="02:00 AM"> 2:00 AM<br>

<input type="checkbox" name="disturb_times:list:date"
value="03:00 AM"> 3:00 AM<br>

<input type="checkbox" name="disturb_times:list:date"
value="04:00 AM"> 4:00 AM<br>

<input type="submit">
</form>
```

By using the *list* and *date* converters together Zope will convert each selected time to a date and then combine all selected dates into a list named *disturb_times*.

A more complex type of form conversion is to convert a series of inputs into records. Records are structures that have attributes. Using records you can combine a number of form inputs into one variable with attributes. The available record converters are:

record
: Converts a variable to a record attribute.

records
: Converts a variable to a record attribute in a list of records.

default
: Provides a default value for a record attribute if the variable is empty.

ignore_empty
: Skips a record attribute if the variable is empty.

Here are some examples of how these converters are used:

```
<form action="processPerson">

First Name <input type="text" name="person.fname:record"><br>
Last Name <input type="text" name="person.lname:record"><br>
Age <input type="text" name="person.age:record:int"><br>

<input type="submit">
</form>
```

This form will call the *processPerson* script with one parameter, *person*. The *person* variable will have *fname*, *lname* and *age* attributes. Here's an example of how you might use the *person* variable in your *processPerson* script:

```
## Script (Python) "processPerson"
##parameters=person
##
" process a person record "
full_name="%s %s" % (person.fname, person.lname)
if person.age < 21:
    return "Sorry, %s. You are not old enough to adopt an aardvark." % full_name
return "Thanks, %s. Your aardvark is on its way." % full_name
```

The *records* converter works like the *record* converter except that it produces a list of records, rather than just one. Here's an example form:

```
<form action="processPeople">

<p>Please, enter information about one or more of your next of
kin.</p>

<p>First Name <input type="text" name="people.fname:records">
Last Name <input type="text" name="people.lname:records"></p>

<p>First Name <input type="text" name="people.fname:records">
Last Name <input type="text" name="people.lname:records"></p>

<p>First Name <input type="text" name="people.fname:records">
Last Name <input type="text" name="people.lname:records"></p>

<input type="submit">
</form>
```

This form will call the *processPeople* script with a variable called *people* that is a list of records. Each record will have *fname* and *lname* attributes.

Another useful parameter conversion uses form variables to rewrite the action of the form. This allows you to submit a form to different scripts depending on how the form is filled out. This is most useful in the case of a form with multiple submit buttons. Zope's action converters are:

action
: Changes the action of the form. This is mostly useful in the case where you have multiple submit buttons on one form. Each button can be assigned to a script that gets called when that button is

clicked to submit the form.

default_action
: Changes the action script of the form when no other *method* converter is found.

Here's an example form that uses action converters:

```
<form action="">

<p>Select one or more employees</p>

<input type="checkbox" name="employees:list" value="Larry"> Larry<br>
<input type="checkbox" name="employees:list" value="Simon"> Simon<br>
<input type="checkbox" name="employees:list" value="Rene"> Rene<br>

<input type="submit" name="fireEmployees:action"
value="Fire!"><br>

<input type="submit" name="promoteEmployees:action"
value="Promote!">

</form>
```

This form will call either the *fireEmployees* or the *promoteEmployees* script depending on which of the two submit buttons is used. Notice also how it builds a list of employees with the *list* converter. Form converters can be very useful when designing Zope applications.

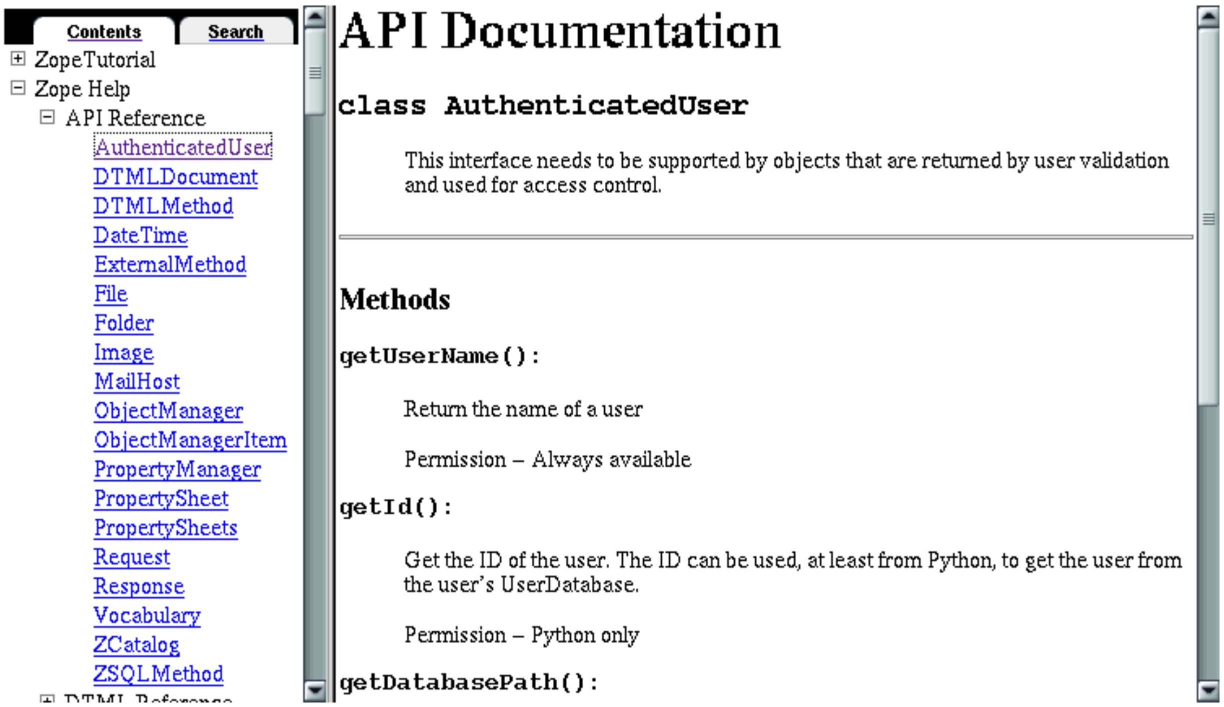
Script Security

All scripts that can be edited through the web are subject to Zope's standard security policies. The only scripts that are not subject to these security restrictions are scripts that must be edited through the filesystem. These unrestricted scripts include Python and Perl *External Methods*.

Chapter 7, "Users and Security" covers security in more detail. You should consult the *Roles of Executable Objects* and *Proxy Roles* sections for more information on how scripts are restricted by Zope security constraints.

The Zope API

One of the main reasons to script Zope is to get convenient access to the Zope API (Application Programmer Interface). The Zope API describes built−in actions that can be called on Zope objects. You can examine the Zope API in the help system, as shown in Figure 8−4.



API Documentation

class AuthenticatedUser

This interface needs to be supported by objects that are returned by user validation and used for access control.

Methods

getUserName():

Return the name of a user

Permission – Always available

getId():

Get the ID of the user. The ID can be used, at least from Python, to get the user from the user's UserDatabase.

Permission – Python only

getDatabasePath():

Figure 8−4 Zope API Documentation

Suppose you'd like to have a script that takes a file you upload from a form and creates a Zope File object in a folder. To do this you need to know a number of Zope API actions. It's easy enough to read files in Python or Perl, but once you have the file you need to know what actions to call to create a new File object in a Folder.

There are many other things that you might like to script using the Zope API. Any management task that you can perform through the web can be scripted using the Zope API. This includes creating, modifying and deleting Zope objects. You can even perform maintenance tasks, like restarting Zope and packing the Zope database.

The Zope API is documented in Appendix B, "API Reference" as well as in the Zope online help. The API documentation shows you which classes inherit from which other classes. For example *Folder* inherits from *ObjectManager*. This means that Folder objects have all the actions listed in the *ObjectManager* section of the API reference.

Using Python−based Scripts

Earlier in this chapter you saw some examples of scripts. Now let's take a look at scripts in more detail.

The Python Language

Python is a high−level, object oriented scripting language. Most of Zope is written in Python. Many folks like Python because of its clarity, simplicity and ability to scale to large projects.

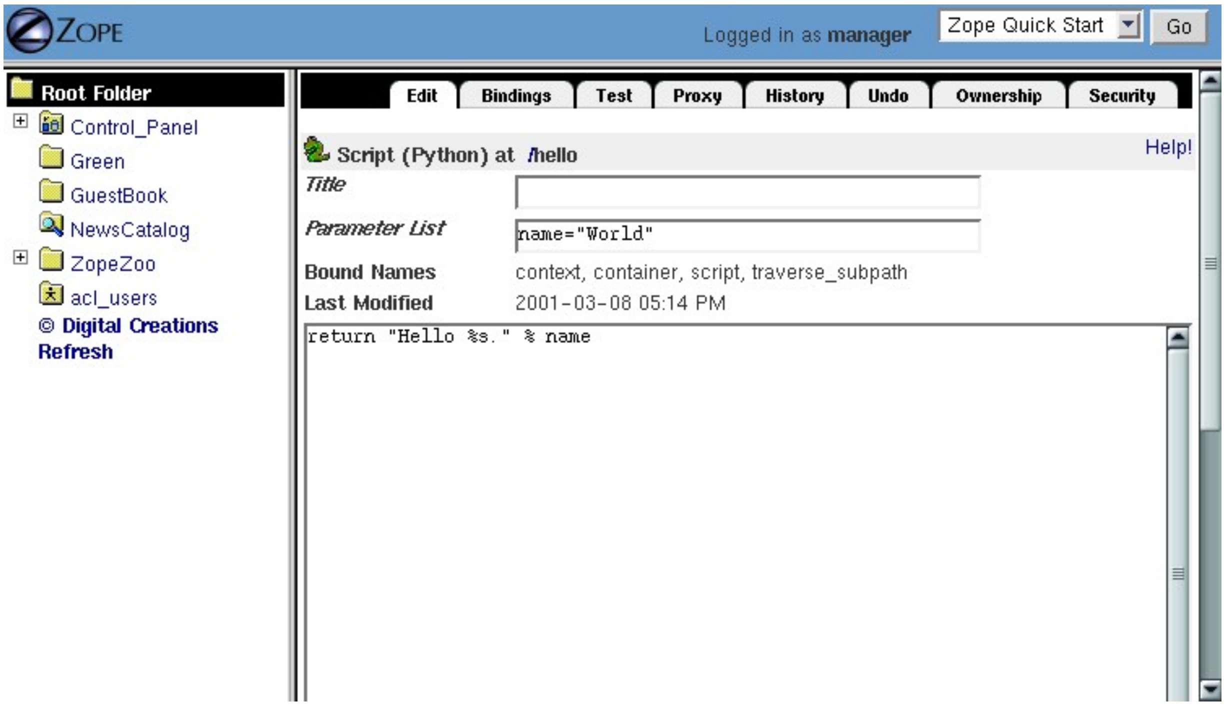
There are many resources available for learning Python. The python.org web site has lots of Python documentation including a tutorial by Python's Creator, Guido van Rossum.

Python comes with a rich set of modules and packages. You can find out more about the Python standard library at the python.org web site.

Another highly respected source for reference material is *Python Essential Reference* by David Beazley published by New Riders.

Creating Python−based Scripts

To create a Python−based Script choose *Script (Python)* from the Product add list. Name the script *hello*, and click the *Add and Edit* button. You should now see the *Edit* view of your script as shown in Figure 8−5.



Title

Parameter List name="World"

Bound Names context, container, script, traverse_subpath

Last Modified 2001-03-08 05:14 PM

```
return "Hello %s." % name
```

Figure 8−5 Script editing view

This screen allows you to control the parameters and body of your script. You can enter your script's parameters in the *parameter list* field. Type the body of your script in the text area at the bottom of the screen.

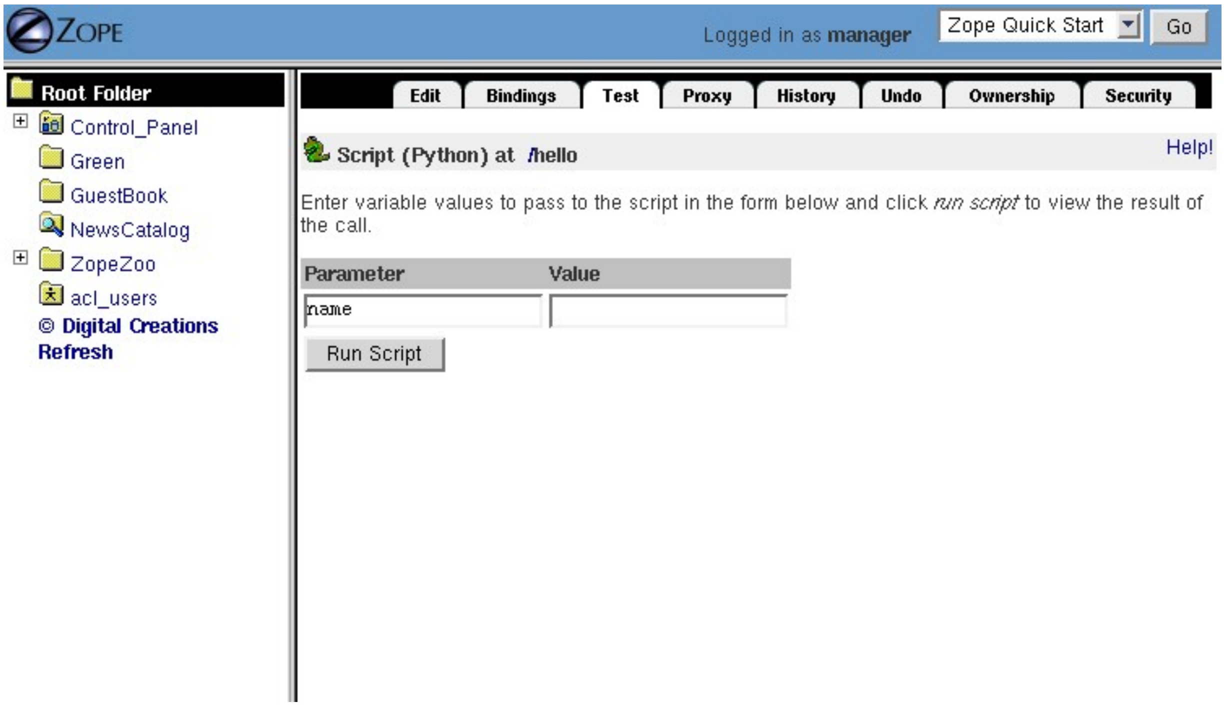
Enter *name="World"* into the *parameter list* field, and type:

```
        return "Hello %s." % name
```

in the body of the script. This is equivalent to this in standard Python syntax:

```
        def hello(name="World"):
            return "Hello %s." % name
```

You can now test this script by going to the *Test* tab as shown in Figure 8−6.



Script (Python) at /hello

Enter variable values to pass to the script in the form below and click *run script* to view the result of the call.

Parameter	Value
name	

Figure 8−6 Testing a Script

Leave the *name* field blank and click the *Run Script* button. Zope should return "Hello World." Now go back and try entering your name in the *Value* field and click the *Run Script* button. Zope should now say hello to you.

Since scripts are called on Zope objects, you can get access to Zope objects via the *context* variable. For example, this script returns the number of objects contained by a given Zope object:

```
## Script (Python) "numberOfObjects
##
return len(context.objectIds())
```

The script calls `context.objectIds()` to find out the number of contained objects. When you call this script on a given Zope object, the context variable is bound to the context object. So if you called this script by visiting the URL *FolderA/FolderB/numberOfObjects* the *context* parameter would refer to the *FolderB* object.

When writing your logic in Python you'll typically want to query Zope objects, call other scripts and return reports. For example, suppose you want to implement a simple workflow system in which various Zope objects are tagged with properties that indicate their status. You might want to produce reports that summarize which objects are in which state. You can use Python to query objects and test their properties. For example, here is a script named *objectsForStatus* with one parameter, *status*:

```
## Script (Python) "objectsForStatus"
##parameters=status
##
"""
Returns all sub−objects that have a given status
property.
"""
results=[]
for object in context.objectValues():
    if object.getProperty('status') == status:
```

```
                results.append(object)
        return results
```

This script loops through an object's sub−objects and returns all the sub−objects that have a *status* property with a given value.

You could then use this script from DTML to email reports. For example:

```
<dtml−sendmail>
To: <dtml−var ResponsiblePerson>
Subject: Pending Objects

These objects are pending and need attention.

<dtml−in expr="objectsForStatus('Pending')">
<dtml−var title_or_id> (<dtml−var absolute_url>)
</dtml−in>
</dtml−sendmail>
```

This example shows how you can use DTML for presentation or report formatting, while Python handles the logic. This is a very important pattern, that you'll see over and over in Zope.

String Processing

One common use for scripts is to do string processing. Python has a number of standard modules for string processing. You cannot do regular expression processing from Python−based Scripts, but you do have access to the *string* module. You have access to the *string* module from DTML as well, but it is much easier to use from Python. Suppose you want to change all the occurrences of a given word in a DTML Document. Here's a script, *replaceWord*, that accepts two arguments, *word* and *replacement*. This will change all the occurrences of a given word in a DTML Document:

```
## Script (Python) "replaceWord"
##parameters=word, replacement
##
"""
Replaces all the occurrences of a word with a
replacement word in the source text of a DTML
Document. Call this script on a DTML Document to use
it.

Note: you'll need permission to edit a document to
call this script on the document.
"""
import string
text=context.document_src()
text=string.replace(text, word, replacement)
context.manage_edit(text, context.title)
```

You can call this script from the web on a DTML Document to change the source of the document. For example, the URL *Swamp/replaceWord?word=Alligatorodile* would call the *replaceWord* script on a document named *Swamp* and would replace all occurrences of the word *Alligator* with *Crocodile*.

The *string* module that you can access via scripts does not have all the features available in the standard Python string module. These limitations are imposed for security reasons. See Appendix A for more information on the *string* module.

One thing that you might be tempted to do with scripts is to use Python to search for objects that contain a given word in their text or as a property. You can do this, but Zope has a much better facility for this kind of work, the *Catalog*. See Chapter 11, "Searching and Categorizing Content" for more information on searching with Catalogs.

Doing Math

Another common use of scripts is to perform mathematical calculations which would be unwieldy from DTML. The *math* and *random* modules give you access from Python to many math functions. These modules are standard Python services as described on the Python.org web site.

math
: Mathematical functions such as *sin* and *cos*.

random
: Pseudo random number generation functions.

One interesting function of the *random* module is the *choice* function that returns a random selection from a sequence of objects. Here's an example of how to use this function in a script called *randomImage*:

```
## Script (Python) "randomImage"
##
"""
When called on a Folder that contains Image objects this
script returns a random image.
"""
import random
return random.choice(context.objectValues('Image'))
```

Suppose you had a Folder named *Images* that contained a number of images. You could display a random image from the folder in DTML like so:

```
<dtml-with Images>
  <dtml-var randomImage>
</dtml-with>
```

This DTML calls the *randomImage* script on the *Images* folder. The result is a HTML *IMG* tag that references a random image in the *Images* Folder.

Binding Variables

A set of special variables is created whenever a Python−based Script is called. These variables, defined on the *Bindings* view, are used by your script to access other Zope objects and scripts.

By default, the names of these binding variables are set to reasonable values and you should not need to change them. They are explained here so that you know how each special variable works, and how you can use these variables in your scripts.

Context
: The *Context* binding defaults to the name *context*. This variable refers to the object that the script is called on.

Container
: The *Container* binding defaults to the name *container*. This variable refers to the folder that the script is defined in.

Script
: The *Script* binding defaults to the name *script*. This variable refers to the script object itself.

Namespace

The *Namespace* binding is left blank by default. This is an advanced variable that you will not need for any of the examples in this book. If your script is called from a DTML Method, and you have chosen a name for this binding, then the named variable contains the DTML namespace explained in Chapter 8, "Variables and Advanced DTML". Also, if this binding is set, the script will search for its parameters in the DTML namespace when called from DTML without explicitly passing any arguments.

Subpath

The *Subpath* binding defaults to the name *traverse_subpath*. This is an advanced variable that you will not need for any of the examples in this book. If your script is traversed, meaning that other path elements follow it in a URL, then those path elements are placed in a list, from left to right, in this variable.

If you edit your scripts via FTP, you'll notice that these bindings are listed in comments at the top of your script files. For example:

```
## Script (Python) "example"
##bind container=container
##bind context=context
##bind namespace=
##bind script=script
##bind subpath=traverse_subpath
##parameters=name, age
##title=
##
return "Hello %s you are %d years old." % (name, age)
```

You can change your script's bindings by changing these comments and then uploading your script.

Print Statement Support

Python−based Scripts have a special facility to help you print information. Normally printed data is sent to standard output and is displayed on the console. This is not practical for a server application like Zope since most of the time you do not have access to the server's console. Scripts allow you to use print anyway and to retrieve what you printed with the special variable *printed*. For example:

```
## Script (Python) "printExample"
##
for word in ('Zope', 'on', 'a', 'rope'):
    print word
return printed
```

This script will return:

```
Zope
on
a
rope
```

The reason that there is a line break in between each word is that Python adds a new line after every string that is printed.

You might want to use the print statement to perform simple debugging in your scripts. For more complex output control you probably should manage things yourself by accumulating data, modifying it and returning it manually rather than relying on the print statement.

Security Restrictions

Scripts are restricted in order to limit their ability to do harm. What could be harmful? In general, scripts keep you from accessing private Zope objects, making harmful changes to Zope objects, hurting the Zope process itself, and accessing the server Zope is running on. These restrictions are implemented through a collection of limits on what your scripts can do.

Loop limits
: Scripts cannot create infinite loops. If your script loops a very large number of times Zope will raise an error. This restriction covers all kinds of loops including *for* and *while* loops. The reason for this restriction is to limit your ability to hang Zope by creating an infinite loop.

Import limits
: Scripts cannot import arbitrary packages and modules. You are limited to importing the *Products.PythonScripts.standard* utility module, the *AccessControl* module, those modules available via DTML (*string*, *random*, *math*, *sequence*), and modules which have been specifically made available to scripts by product authors. See Appendix B, "API Reference" for more information on these modules. If you want to be able to import any Python module, use an External Method, as described later in the chapter.

Access limits
: You are restricted by standard Zope security policies when accessing objects. In other words the user executing the script is checked for authorization when accessing objects. As with all executable objects you can modify the effective roles a user has when calling a script using *Proxy Roles* (see Chapter 7, "Users and Security", for more information.) In addition, you cannot access objects whose names begin with underscore, since Zope considers these objects to be private.

Writing limits
: In general you cannot change Zope object attributes using scripts. You should call scripts on Zope objects to change them, rather than directly changing instance attributes.

Despite these limits, a determined user could use large amounts of CPU time and memory using Python−based Scripts. So malicious scripts could constitute a kind of denial of service attack by using lots of resources. These are difficult problems to solve and DTML suffers from the same potential for abuse. As with DTML, you probably shouldn't grant access to scripts to untrusted people.

Built−in Functions

Python−based Scripts give you a slightly different menu of built−ins than you find in normal Python. Most of the changes are designed to keep you from performing unsafe actions. For example, the *open* function is not available, which keeps you from being able to access the filesystem. To partially make up for some missing built−ins a few extra functions are available.

These restricted built−ins work the same as standard Python built−ins: *None*, *abs*, *apply*, *callable*, *chr*, *cmp*, *complex*, *delattr*, *divmod*, *filter*, *float*, *getattr*, *hash*, *hex*, *int*, *isinstance*, *issubclass*, *list*, *len*, *long*, *map*, *max*, *min*, *oct*, *ord*, *repr*, *round*, *setattr*, *str*, *tuple*. For more information on what these built−ins do, see the online Python Documentation.

The *range* and *pow* functions are available and work the same way they do in standard Python; however, they are limited to keep them from generating very large numbers and sequences. This limitation helps protect against denial of service attacks as described previously.

In addition, these DTML utility functions are available: *DateTime*, and *test*. See Appendix A, "DTML Reference" for more information on these functions.

Finally to make up for the lack of a *type* function, there is a *same_type* function that compares the type of two or more objects, returning true if they are of the same type. So instead of saying:

```
if type(foo) == type([]):
    return "foo is a list"
```

to check if `foo` is a list, you would instead use the *same_type* function to check this:

```
if same_type(foo, []):
    return "foo is a list"
```

Now let's take a look at *External Methods* which provide more power and less restrictions than Python−based Scripts.

Using External Methods

Sometimes the security constraints imposed by scripts get in your way. For example, you might want to read files from disk, or access the network, or use some advanced libraries for things like regular expressions or image processing. In these cases you'll want to use *External Methods*.

To create and edit External Methods you need access to the filesystem. This makes editing these scripts more cumbersome since you can't edit them right in your web browser. However requiring access to the server's filesystem provides an important security control. If a user has access to a servers filesystem they already have the ability to harm Zope. So by requiring that unrestricted scripts be edited on the filesystem Zope ensures that only people who are already trusted have access.

Unrestricted scripts are created and edited in files on the Zope server in the *Extensions* directory. This directory is located in the top−level Zope directory. Alternately you can create and edit unrestricted scripts in an *Extensions* directory inside an installed Zope product directory.

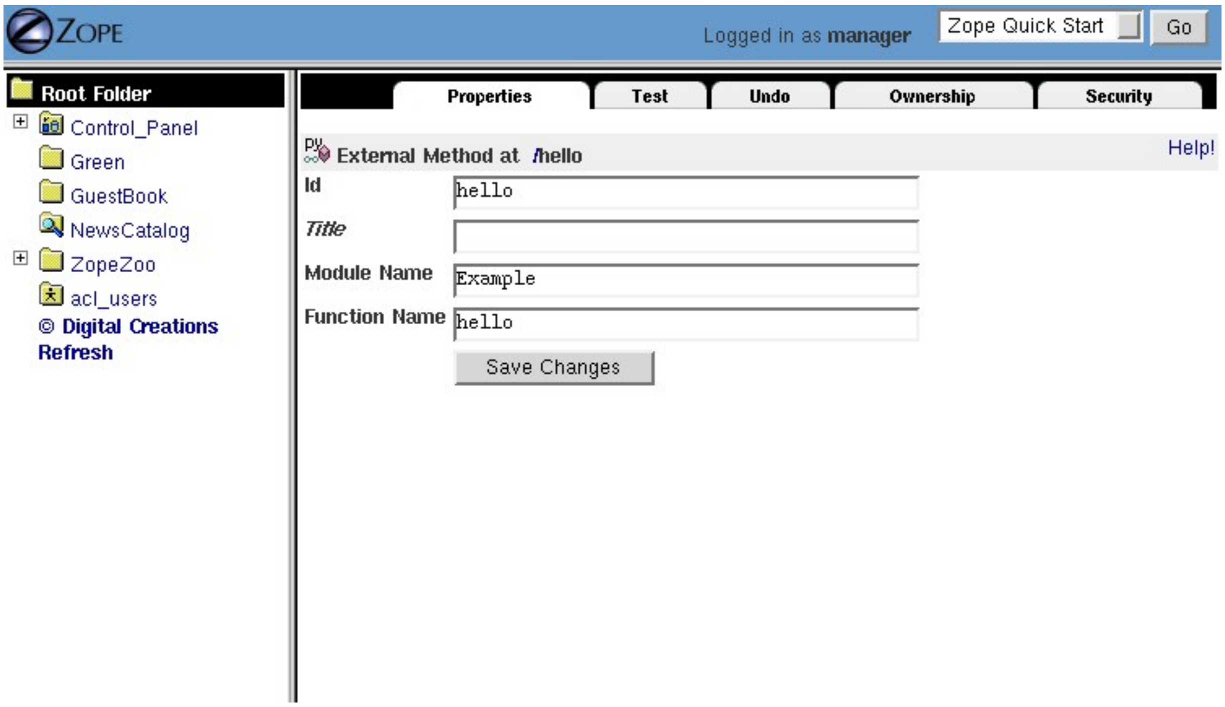
Create a file named *Example.py* in the Zope *Extensions* directory on your server. In the *Example.py* file, enter the following code:

```
def hello(name="World"):
    return "Hello %s." % name
```

You've created a Python function in a Python module. Now let's use this function in the External Method.

You manage External Methods the same way you manage restricted scripts with the exception that you cannot edit the script itself through the web. Instead of editing code you must tell Zope where to find your code on the filesystem. You do this by specifying the name of your Python file and the name of the function within the module.

To create an External Method choose *External Method* from the product add list. You will be taken to an add form where you must provide an id. Type "hello" into the *Id* field and "hello" in the *Function name* field and "Example" in the *Module name* field and click the *Add* button. You should now see a new External Method object in your folder. Click on it. You should be taken to the *Properties* view of your new External Method as shown in Figure 8−7.



External Method at /hello

Id	hello
Title	
Module Name	Example
Function Name	hello

Figure 8−7 External Method Properties view

Now test your new script by going to the *Test* view. You should see a greeting. You can pass different names to the script by specifying them in the URL. For example, *hello?name=Spanish+Inquisition*.

This example is exactly the same as the hello world example that you saw for using scripts. In fact for simple string processing tasks like this restricted scripts offer a better solution since they are easier to work with.

The main reasons to use an unrestricted script are to access the filesystem or network or to use Python packages that are not available to restricted scripts.

Here's an example External Method that uses the Python Imaging Library (PIL) to create a thumbnail version of an existing Image object in a Folder. Enter the following code in a file named *Thumbnail.py* in the *Extensions* directory:

```
def makeThumbnail(self, original_id, size=200):
    """
    Makes a thumbnail image given an image Id when called on a Zope
    folder.

    The thumbnail is a Zope image object that is a small JPG
    representation of the original image. The thumbnail has a
    'original_id' property set to the id of the full size image
    object.
    """

    from PIL import Image
    from StringIO import StringIO
    import os.path

    # create a thumbnail image file
    original_image=getattr(self, original_id)
    original_file=StringIO(str(original_image.data))
    image=Image.open(original_file)
    image=image.convert('RGB')
```

```
          image.thumbnail((size,size))
          thumbnail_file=StringIO()
          image.save(thumbnail_file, "JPEG")
          thumbnail_file.seek(0)

          # create an id for the thumbnail
          path, ext=os.path.splitext(original_id)
          thumbnail_id=path + '.thumb.jpg'

          # if there's and old thumbnail, delete it
          if thumbnail_id in self.objectIds():
              self.manage_delObjects([thumbnail_id])

          # create the Zope image object
          self.manage_addProduct['OFSP'].manage_addImage(thumbnail_id,
                                                         thumbnail_file,
                                                         'thumbnail image')
          thumbnail_image=getattr(self, thumbnail_id)

          # set the 'originial_id' property
          thumbnail_image.manage_addProperty('original_id', original_id, 'string')
```

You must have PIL installed for this example to work. See the PythonWorks website for more information on PIL. To use this code create an External Method named *makeThumbnail* that uses the *makeThumbnail* function in the *Thumbnail* module.

Now you have a method that will create a thumbnail image. You can call it on a Folder with a URL like *ImageFolder/makeThumbnail?original_id=Horse.gif* This would create a thumbnail image named `Horse.thumb.jpg`.

You can use a script to loop through all the images in a folder and create thumbnail images for them. Create a script named *makeThumbnails*:

```
      ## Script (Python) "makeThumbnails"
      ##
      for image_id in context.objectIds('Image'):
          context.makeThumbnail(image_id)
```

This will loop through all the images in a folder and create a thumbnail for each one.

Now call this script on a folder with images in it. It will create a thumbnail image for each contained image. Try calling the *makeThumbnails* script on the folder again and you'll notice it created thumbnails of your thumbnails. This is no good. You need to change the *makeThumbnails* script to recognize existing thumbnail images and not make thumbnails of them. Since all thumbnail images have an *original_id* property you can check for that property as a way of distinguishing between thumbnails and normal images:

```
      ## Script (Python) "makeThumbnails"
      ##
      for image in context.objectValues('Image'):
          if not image.hasProperty('original_id'):
              context.makeThumbnail(image.getId())
```

Delete all the thumbnail images in your folder and try calling your updated *makeThumbnails* script on the folder. It seems to work correctly now.

Now with a little DTML you can glue your script and External Method together. Create a DTML Method called *displayThumbnails*:

```
<dtml-var standard_html_header>

<dtml-if updateThumbnails>
  <dtml-call makeThumbnails>
</dtml-if>

<h2>Thumbnails</h2>

<table><tr valign="top">

<dtml-in expr="objectValues('Image')">
  <dtml-if original_id>
    <td>
      <a href="&dtml-original_id;"><dtml-var sequence-item></a><br>
      <dtml-var original_id>
    </td>
  </dtml-if>
</dtml-in>

</tr></table>

<form>
<input type="submit" name="updateThumbnails" value="Update Thumbnails">
</form>

<dtml-var standard_html_footer>
```

When you call this DTML Method on a folder it will loop through all the images in the folder and display all the thumbnail images and link them to the originals as shown in Figure 8−8.

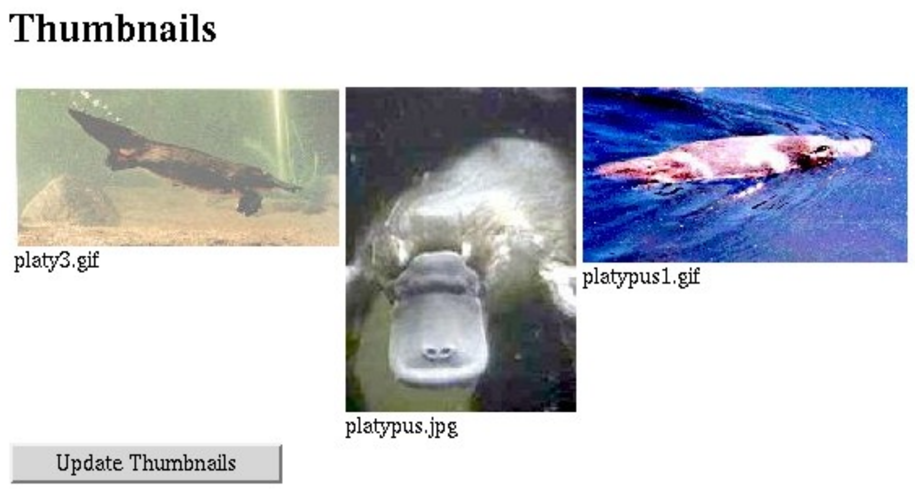


Figure 8−8 Displaying thumbnail images

This DTML Method also includes a form that allows you to update the thumbnail images. If you add, delete or change the images in your folder you can use this form to update your thumbnails.

This example shows how to use scripts, External Methods and DTML together. Python takes care of the logic while the DTML handles presentation. Your External Methods handle external packages while your scripts do simple processing of Zope objects.

Processing XML with External Methods

You can use External Methods to do darn near anything. One interesting thing that you can do is to communicate using XML. You can generate and process XML with External Methods.

Zope already understands some kinds of XML messages such as XML−RPC and WebDAV. As you create web applications that communicate with other systems you may want to have the ability to receive XML messages. You can receive XML a number of ways: you can read XML files from the file system or over the network, or you can define scripts that take XML arguments which can be called by remote systems.

Once you have received an XML message you must process the XML to find out what it means and how to act on it. Let's take a quick look at how you might parse XML manually using Python. Suppose you want to connect your web application to a Jabber chat server. You might want to allow users to message you and receive dynamic responses based on the status of your web application. For example suppose you want to allow users to check the status of animals using instant messaging. Your application should respond to XML instant messages like this:

```
<message to="cage_monitor@zopezoo.org" from="user@host.com">
  <body>monkey food status</body>
</message>
```

You could scan the body of the message for commands, call a script and return responses like this:

```
<message to="user@host.com" from="cage_monitor@zopezoo.org">
  <body>Monkeys were last fed at 3:15</body>
</message>
```

Here is a sketch of how you could implement this XML messaging facility in your web application using an External Method:

```
# Uses Python 2.x standard xml processing packages.  See
# http://www.python.org/doc/current/lib/module-xml.sax.html for
# information about Python's SAX (Simple API for XML) support If
# you are using Python 1.5.2 you can get the PyXML package. See
# http://pyxml.sourceforge.net for more information about PyXML.

from xml.sax import parseString
from xml.sax.handler import ContentHandler

class MessageHandler(ContentHandler):
    """
    SAX message handler class

    Extracts a message's to, from, and body
    """

    inbody=0
    body=""

    def startElement(self, name, attrs):
        if name=="message":
            self.recipient=attrs['to']
            self.sender=attrs['from']
        elif name=="body":
            self.inbody=1
    def endElement(self, name):
        if name=="body":
            self.inbody=0

    def characters(self, content):
        if self.inbody:
            self.body=self.body + content

def receiveMessage(self, message):
    """
    Called by a Jabber server
    """
    handler=MessageHandler()
    parseString(message, handler)

    # call a script that returns a response string
    # given a message body string
    response_body=self.getResponse(handler.body)

    # create a response XML message
    response_message="""
      <message to="%s" from="%s">
        <body>%s</body>
      </message>""" % (handler.sender, handler.recipient, response_body)

    # return it to the server
    return response_message
```

The *receiveMessage* External Method uses Python's SAX (Simple API for XML) package to parse the XML message. The *MessageHandler* class receives callbacks as Python parses the message. The handler saves information its interested in. The External Method uses the handler class by creating an instance of it, and passing it to the *parseString* function. It then figures out a response message by calling *getResponse* with the message body. The *getResponse* script (which is not shown here) presumably scans the body for commands, queries the web applications state and returns some response. The *receiveMessage* method then creates an XML message using response and the sender information and returns it.

The remote server would use this External Method by calling the *receiveMessage* method using the standard HTTP POST command. Voila, you've implemented a custom XML chat server that runs over HTTP.

External Method Gotchas

While you are essentially unrestricted in what you can do in an External Method, there are still some things that are hard to do.

While your Python code can do as it pleases if you want to work with the Zope framework you need to respect its rules. While programming with the Zope framework is too advanced a topic to cover here, there are a few things that should be aware of.

Problems can occur if you hand instances of your own classes to Zope and expect them to work like Zope objects. For example, you cannot define a class in an External Method script file and assign it as an attribute of a Zope object. This causes problems with Zope's persistence machinery. You also cannot easily hand instances of your own classes over to DTML or scripts. The issue here is that your instances won't have Zope security information. You can define and use your own classes and instances to your heart's delight, just don't expect Zope to use them directly. Limit yourself to returning simple Python structures like strings, dictionaries and lists or Zope objects.

Using Perl−based Scripts

Perl−based Scripts allow you to script Zope in Perl. If you love Perl and don't want to learn Python to use Zope, these scripts are for you. Using Perl−based Scripts you can use all your favorite Perl modules and treat Zope like a collection of Perl objects.

The Perl Language

Perl is a high−level scripting language like Python. From a broad perspective, Perl and Python are very similar languages, they have similar primitive data constructs and employ similar programming constructs.

Perl is a popular language for Internet scripting. In the early days of CGI scripting, Perl and CGI were practically synonymous. Perl continues to be the dominant Internet scripting language.

Perl has a very rich collection of modules for tackling almost any computing task. CPAN (Comprehensive Perl Archive Network) is the authoritative guide to Perl resources.

Perl−based Zope scripts are available for download from ActiveState. Perl−based scripts require you to have Perl installed, and a few other packages, and how to install these things is beyond the scope of this book. See the documentation that comes with Perl−based scripts from the above URL. There is also more information provided by Andy McKay available on Zope.org.

Creating Perl−based Scripts

Perl−based Scripts are quite similar to Python−based Scripts. Both have access to Zope objects and are called in similar ways. Here's the Perl hello world program:

```
my $name=shift;
return "Hello $name.";
```

Let's take a look at a more complex example script by Monty Taylor. It uses the `LWP::UserAgent` package to retrieve the URL of the daily Dilbert comic from the network. Create a Perl−based Script named *get_dilbert_url* with this code:

```
use LWP::UserAgent;

my $ua = LWP::UserAgent->new;

# retrieve the Dilbert page
my $request = HTTP::Request->new('GET','http://www.dilbert.com');
my $response = $ua->request($request);

# look for the image URL in the HTML
my $content = $response->content;
$content =~ m,(/comics/dilbert/archive/images/[^"]*),s;

# return the URL
return $content
```

You can display the daily Dilbert comic by calling this script from DTML by calling the script inside an HTML *IMG* tag:

```
<img src="&dtml-get_dilbert_url;">
```

However there is a problem with this code. Each time you display the cartoon, Zope has to make a network connection. This is inefficient and wasteful. You'd do much better to only figure out the Dilbert URL once a day.

Here's a script *cached_dilbert_url* that improves the situation by keeping track of when it last fetched the Dilbert URL with a *dilbert_url_date* property:

```
my $context=shift;
my $date=$context->getProperty('dilbert_url_date');

if ($date==null or $now-$date > 1){
    my $url=$context->get_dilbert_url();
    $context->manage_changeProperties(
      dilbert_url => $url
      dilbert_url_time => $now
    );
}
return $context->getProperty('dilbert_url');
```

This script uses two properties, *dilbert_url* and *dilbert_url_date*. If the URL gets too old, a new one is fetched. You can use this script from DTML just like the original script:

```
<img src="&dtml-cached_dilbert_url;">
```

You can use Perl and DTML together to control your logic and your presentation.

Perl−based Script Security

Like DTML and Python−based Scripts, Perl−based Scripts constrain you in the Zope security system from doing anything that you are not allowed to do. Script security is similar in both languages, but there are some Perl specific constraints.

First, the security system does not allow you to *eval* an expression in Perl. For example, consider this script:

```
my $context = shift;
my $input = shift;

eval $input
```

This code takes an argument and evaluates it in Perl. This means you could call this script from, say an HTML form, and evaluate the contents of one of the form elements. This is not allowed since the form element could contain malicious code.

Perl−based Scripts also cannot assign new variables to any object other than local variables that you declare with *my*.

DTML versus Python versus Perl

Zope gives you many ways to script. For small scripting tasks the choice of Python, Perl or DTML probably doesn't make a big difference. For larger, logic−oriented tasks you should use Python or Perl. You should choose the language you are most comfortable with. Of course, your boss may want to have some say in the matter too.

Just for comparison sake here is a simple script suggested by Gisle Aas, the author of Perl−based Scripts, in three different languages.

In DTML:

```
<dtml-in objectValues>
  <dtml-var getId>: <dtml-var sequence-item>
</dtml-in>
done
```

In Python:

```
for item in context.objectValues():
    print "%s: %s" % (item.getId(), item)
print "done"
return printed
```

In Perl:

```
my $context = shift;
my @res;

for ($context->objectValues()) {
    push(@res, join(": ", $_->getId(), $_));
}
join("\n", @res, "done");
```

Despite the fact that Zope is implemented in Python, it follows the Perl philosophy that there's more than one way to do it.

Remote Scripting and Network Services

Web servers are used to serve content to software clients; usually people using web browser software. The software client can also be another computer that is using your web server to access some kind of service.

Because Zope exposes objects and scripts on the web, it can be used to provide a powerful, well organized, secure web API to other remote network application clients.

There are two common ways to remotely script Zope. The first way is using a simple remote procedure call protocol called *XML−RPC*. XML−RPC is used to execute a procedure on a remote machine and get a result on the local machine. XML−RPC is designed to be language neutral, and in this chapter you'll see examples in Python, Perl and Java.

The second common way to remotely script Zope is with any HTTP client that can be automated with a script. Many language libraries come with simple scriptable HTTP clients and there are many programs that let you you script HTTP from the command line.

Using XML−RPC

XML−RPC is a simple remote procedure call mechanism that works over HTTP and uses XML to encode information. XML−RPC clients have been implemented for many languages including Python, Perl, Java, JavaScript, and TCL.

In−depth information on XML−RPC can be found at the XML−RPC website.

All Zope scripts that can be called from URLs can be called via XML−RPC. Basically XML−RPC provides a system to marshal arguments to scripts that can be called from the web. As you saw earlier in the chapter Zope provides its own marshaling controls that you can use from HTTP. XML−RPC and Zope's own marshaling accomplish much the same thing. The advantage of XML−RPC marshaling is that it is a reasonably supported standard that also supports marshaling of return values as well as argument values.

Here's a fanciful example that shows you how to remotely script a mass firing of janitors using XML−RPC.

Here's the code in Python:

```
import xmlrpclib

server = xmlrpclib.Server('http://www.zopezoo.org/')
for employeeID in server.JanitorialDepartment.personnel():
    server.fireEmployee(employee)
```

In Perl:

```
use Frontier::Client;

$server = Frontier::Client->new(url => "http://www.zopezoo.org/");

$employees = $server->call("JanitorialDepartment.personnel");
foreach $employee ( @$employees ) {

  $server->call("fireEmployee",$server->string($employee));

}
```

In Java:

```
try {
    XmlRpcClient server = new XmlRpcClient("http://www.zopezoo.org/");
    Vector employees = (Vector) server.execute("JanitorialDepartment.personnel");

    int num = employees.size();
    for (int i = 0; i < num; i++) {
        Vector args = new Vector(employees.subList(i, i+1));
        server.execute("fireEmployee", args);
    }

} catch (XmlRpcException ex) {
    ex.printStackTrace();
} catch (IOException ioex) {
    ex.printStackTrace();
}
```

Actually the above example will probably not run correctly, since you will most likely want to protect the *fireEmployee* script. This brings up the issue of security with XML−RPC. XML−RPC does not have any security provisions of its own; however, since it runs over HTTP it can leverage existing HTTP security controls. In fact Zope treats an XML−RPC request exactly like a normal HTTP request with respect to security controls. This means that you must provide authentication in your XML−RPC request for Zope to grant you access to protected scripts. The Python client at the time of this writing does not support control of HTTP Authorization headers. However it is a fairly trivial addition. For example, an article on XML.com Internet Scripting: Zope and XML−RPC includes a patch to Python's XML−RPC support showing how to add HTTP authorization headers to your XML−RPC client.

Remote Scripting with HTTP

Any HTTP client can be used for remotely scripting Zope.

On Unix systems you have a number of tools at your disposal for remotely scripting Zope. One simple example is to use *wget* to call Zope script URLs and use *cron* to schedule the script calls. For example, suppose you have a Zope script that feeds the lions and you'd like to call it every morning. You can use *wget* to call the script like so:

```
$ wget --spider http://www.zopezope.org/Lions/feed
```

The *spider* option tells *wget* not to save the response as a file. Suppose that your script is protected and requires authorization. You can pass your user name and password with *wget* to access protected scripts:

```
$ wget --spider --http_user=ZooKeeper --http_pass=SecretPhrase http://www.zopezope.org/Li
```

Now let's use *cron* to call this command every morning at 8am. Edit your crontab file with the *crontab* command:

```
$ crontab -e
```

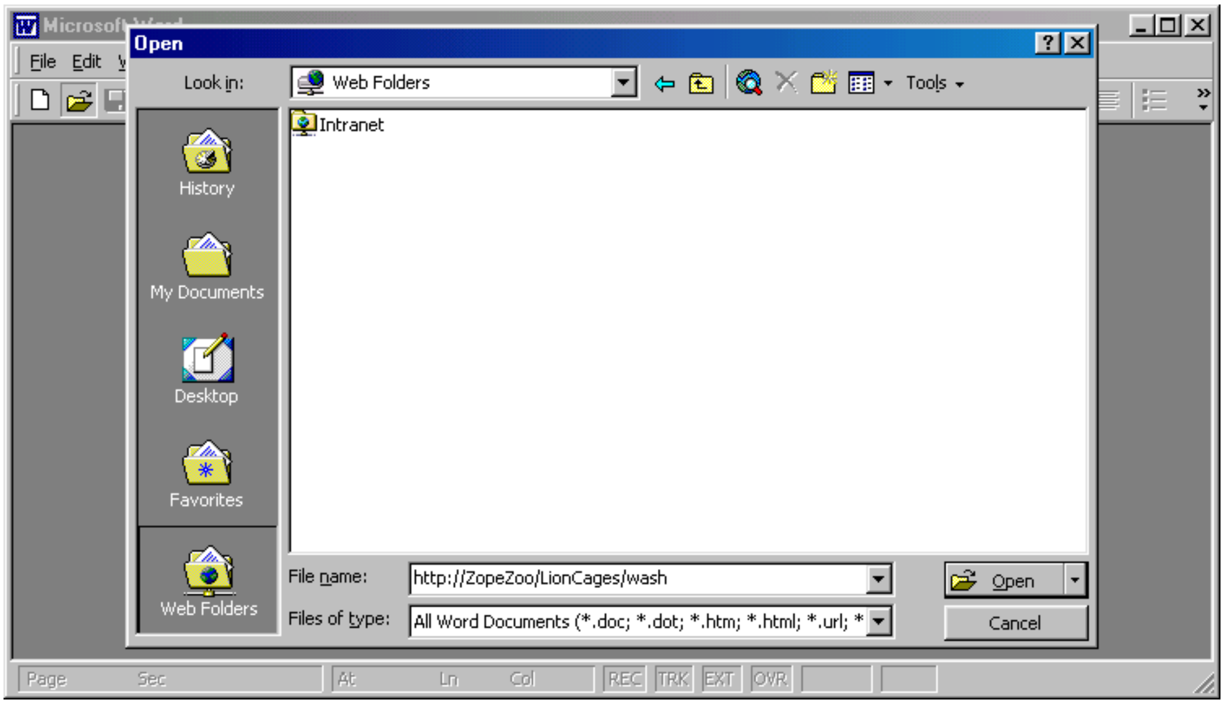
Then add a line to call wget every day at 8 am:

```
0 8 * * * wget -v --spider --http_user=ZooKeeper --http_pass=SecretPhrase http://www.zope
```

The only difference between using *cron* and calling *wget* manually is that you should use the *v* switch when using *cron* since you don't care about output of the *wget* command.

For our final example let's get really perverse. Since networking is built into so many different systems, it's easy to find an unlikely candidate to script Zope. If you had an Internet−enabled toaster you would probably be able to script Zope with it. Let's take Microsoft Word as our example Zope client. All that's necessary is to get Word to agree to tickle a URL.

The easiest way to script Zope with Word is to tell word to open a document and then type a Zope script URL as the file name as shown in [Figure 8−9](#).



Look in: Web Folders

File name: http://ZopeZoo/LionCages/wash

Files of type: All Word Documents (*.doc; *.dot; *.htm; *.html; *.url; *

Figure 8−9 Calling a URL with Microsoft Word

Word will then load the URL and return the results of calling the Zope script. Despite the fact that Word doesn't let you POST arguments this way, you can pass GET arguments by entering them as part of the URL.

You can even control this behavior using Word's built−in Visual Basic scripting. For example, here's a fragment of Visual Basic that tells Word to open a new document using a Zope script URL:

```
Documents.Open FileName:="http://www.zopezoo.org/LionCages/wash?use_soap=1&water_temp=hot
```

You could use Visual Basic to call Zope script URLs in many different ways.

Zope's URL to script call translation is the key to remote scripting. Since you can control Zope so easily with simple URLs you can easy script Zope with almost any network−aware system.

Conclusion

Zope provides scripting with Python and Perl. With scripts you can control Zope objects and glue together your application's logic, data, and presentation. You can also perform serious programming tasks such as image processing and XML parsing.

In the next chapter you'll learn about ZCatalog, Zope's built−in search engine.

Chapter 11: Searching and Categorizing Content

The Catalog is Zope's built in search engine. It allows you to categorize and search all kinds of Zope objects. You can also use it to search external data such as relational data, files, and remote web pages. In addition to searching you can use the Catalog to organize collections of objects.

The Catalog supports a rich query interface. You can perform full text searching, and can search multiple indexes at once. In addition, the catalog keeps track of meta−data about indexed objects. Here are the two most common ZCatalog usage patterns:

Mass Cataloging
: Cataloging a large collection of objects all at once.

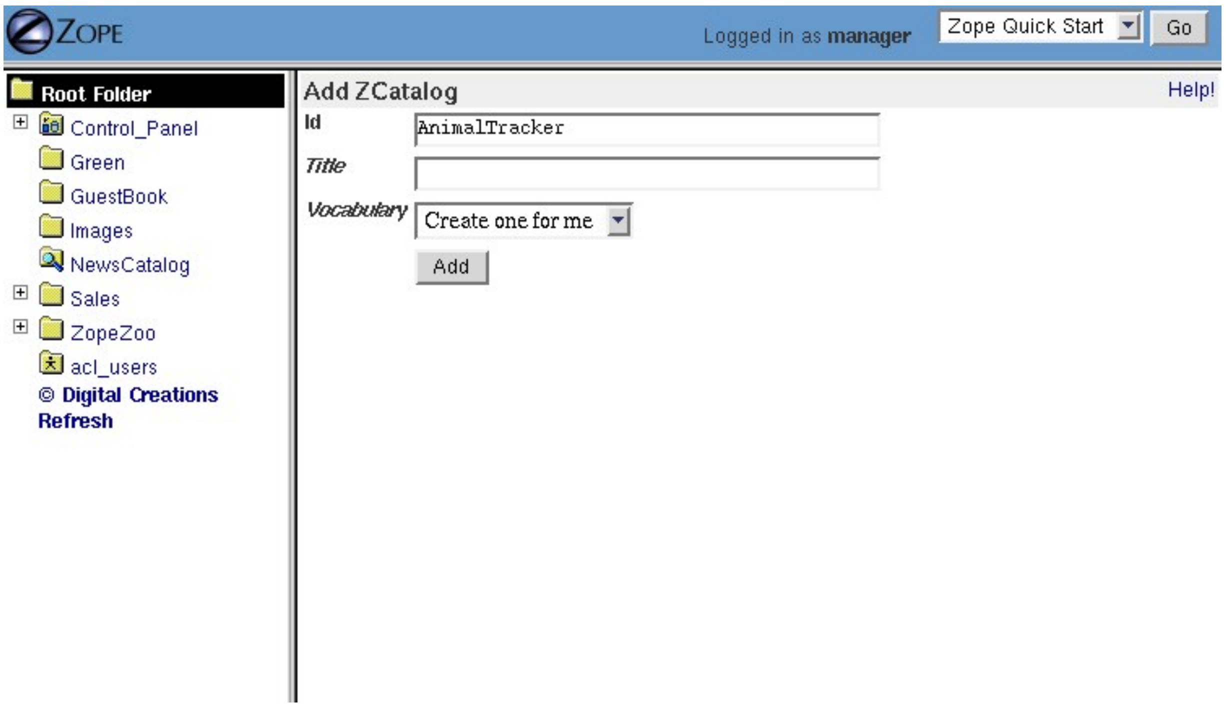
Automatic Cataloging
: Cataloging objects as they are created and tracking changes made to them.

Getting started with Mass Cataloging

Let's take a look at how to use the catalog to search documents. Cataloging a bunch of objects all at once is called *mass cataloging*. Mass cataloging involves three steps:

- Creating a ZCatalog
- Finding objects and cataloging them
- Creating a web interface to search the catalog.

Choose *ZCatalog* from the product add list to create a ZCatalog object. This takes you to the ZCatalog add form, as shown in Figure 9−1.



Add ZCatalog

Id	AnimalTracker
Title	
Vocabulary	Create one for me

Add

Figure 9−1 ZCatalog add form

The Add form asks you for an *Id* and a *Title*. The third form element is the *Vocabulary* select box. For now, leave this box on "Create one for me". Give your ZCatalog the Id "AnimalTracker" and click *Add* to create your new catalog. The Catalog icon looks like a folder with a small magnifying glass on it. Select the *AnimalTracker* icon to see the *Contents* view of the Catalog.

A ZCatalog looks a lot like a folder, but it has a few more tabs. Six tabs on the ZCatalog are the exact same six tabs you find on a standard folder. ZCatalog have the following views: *Contents*, *Catalog*, *Properties*, *Indexes*, *MetaData*, *Find Objects*, *Advanced*, *Undo*, *Security*, and *Ownership*. When you click on a ZCatalog, you are on the *Contents* view. Here, you can add new objects and the ZCatalog will contain them just as any folder does. You should note that containment does not imply that the object is searchable.

Now that you have created a ZCatalog, you can move onto the next step, finding objects and cataloging them. Suppose you have a zoo site with information about animals. To work with these examples, create two DTML Documents that contain information about reptiles and amphibians:

Title: Chilean four−eyed frog
: The Chilean four−eyed frog has a bright pair of spots on its rump that look like enormous eyes. When seated, the frog's thighs conceal these eyespots. When predators approach, the frog lowers its head and lifts its rump, creating a much larger and more intimidating head. Frogs are amphibians.

Title: Carpet python
: *Morelia spilotes variegata* averages 2.4 meters in length. It is a medium−sized python with black−to−gray patterns of blotches, crossbands, stripes, or a combination of these markings on a light yellowish−to−dark brown background. Snakes are reptiles.

Visitors to your Zoo want to be able to search for information on the Zoo's animals. Eager herpetologists want to know if you have their favorite snake, so you should provide them with the ability to search for certain words and show all the documents that contain those words. Searching is one of the most useful and common web activities.

The *AnimalTracker* ZCatalog you created can catalog all of the documents in your Zope site and let your users search for specific words. To catalog your documents, go to the *AnimalTracker* ZCatalog and click on the *Find Objects* tab.

In this view, you tell the ZCatalog what kind of objects you are interested in. You want to catalog all DTML Documents so select *DTML Document* from the *Find objects of type* multiple selection and click *Find and Catalog*.

The ZCatalog will now start from the folder where it is located and search for all DTML Documents. It will search the folder and then descend down into all of the sub−folders and their sub−folders. If you have lots and lots of objects, this may take a long time to complete, so be patient.

After a period of time, the Catalog will take you to the *Catalog* view automatically, with a status message telling you what it just did.

Below the status information is a list of objects that are cataloged, they are all DTML Documents. To confirm that these are the objects you are interested in, you can click on them to visit them.

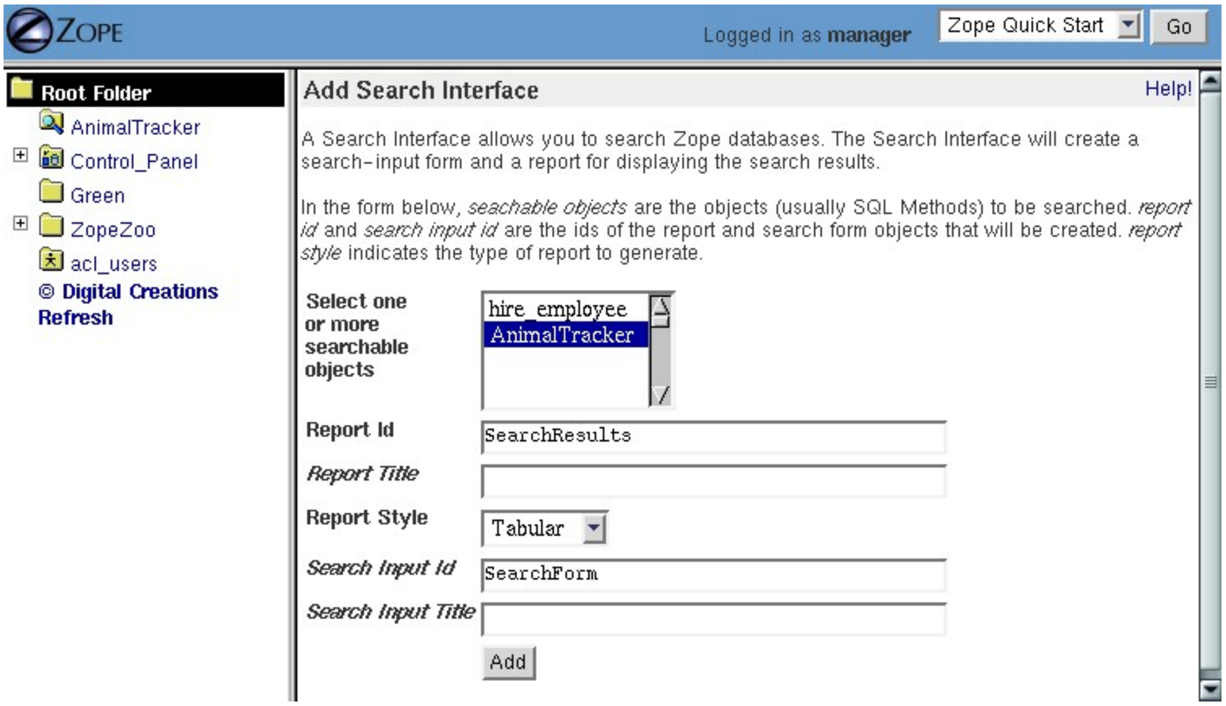
You have completed the first step of searching your objects, cataloging them into a ZCatalog. Now your documents are in the ZCatalog's database. Now you can move onto the third step, creating a web page and result form to query the ZCatalog.

Below the status information is a list of objects that are cataloged. They are all DTML Documents. To confirm that these are the objects you are interested in, you can click on them to visit them.

You have completed the first step of searching your objects, cataloging them into a ZCatalog. Now your documents are in the ZCatalog's database. Now you can move onto the third step, creating a web page and result form to query the ZCatalog.

Search and Report Forms

To create search and report forms, make sure you are inside the *AnimalTracker* catalog and select *Z Search Interface* from the add list. Select the *AnimalTracker* ZCatalog as the searchable object, as shown in Figure 9−2.



Add Search Interface

A Search Interface allows you to search Zope databases. The Search Interface will create a search-input form and a report for displaying the search results.

In the form below, *seachable objects* are the objects (usually SQL Methods) to be searched. *report id* and *search input id* are the ids of the report and search form objects that will be created. *report style* indicates the type of report to generate.

Select one or more searchable objects	hire_employee, AnimalTracker (selected)
Report Id	SearchResults
Report Title	
Report Style	Tabular
Search Input Id	SearchForm
Search Input Title	

Figure 9−2 Creating a search form for a ZCatalog

Name the *Report Id* "SearchResults" and the *Search Input Id* "SearchForm" and click *Add*. This will create two new DTML Methods in the *AnimalTracker* ZCatalog named *SeachForm* and *SearchResults*.

These objects are *contained in* the ZCatalog, but they are not *cataloged by* the ZCatalog. The *AnimalTracker* has only cataloged DTML Documents. The search Form and Report methods are just a user interface to search the animal documents in the Catalog. You can verify this by noting that the search and report forms are not listed in the *Cataloged Objects* tab.

To search the *AnimalTracker* ZCatalog, select the *SearchForm* method and click on its *View* tab. This form has a number of elements on it. There is one search element for each index in the ZCatalog. Indexes are explained further in the next section. For now, you want to use the *PrincipiaSearchSource* form element. You can leave all the other form elements blank.

By typing words into the *PrincipiaSearchSource* form element you can search all of the documents cataloged by the *AnimalTracker* ZCatalog. For example, type in the word "Reptiles". The *AnimalTracker* ZCatalog will be searched and return a simple table of objects that have the word "Reptiles" in them. The search results should include the carpet python. You can also try specifying multiple search terms like "reptile amphibian". Search results for this query should include both the Chilean four−eyed Frog and the carpet python. Congratulations, you have successfully created a catalog, cataloged content into it and searched it through the web.

Configuring Catalogs

The Catalog is capable of much more powerful and complex searches than the one you just performed. Let's take a look at how the Catalog stores information. This will help you tailor your catalogs to provide the sort of searching you want.

Defining Indexes

ZCatalogs store information about objects and their contents in fast databases called *indexes*. Indexes can store and retrieve large volumes of information very quickly. You can create different kinds of indexes that remember different kinds of information about your objects. For example, you could have one index that remembers the text content of DTML Documents, and another index that remembers any objects that have a specific property.

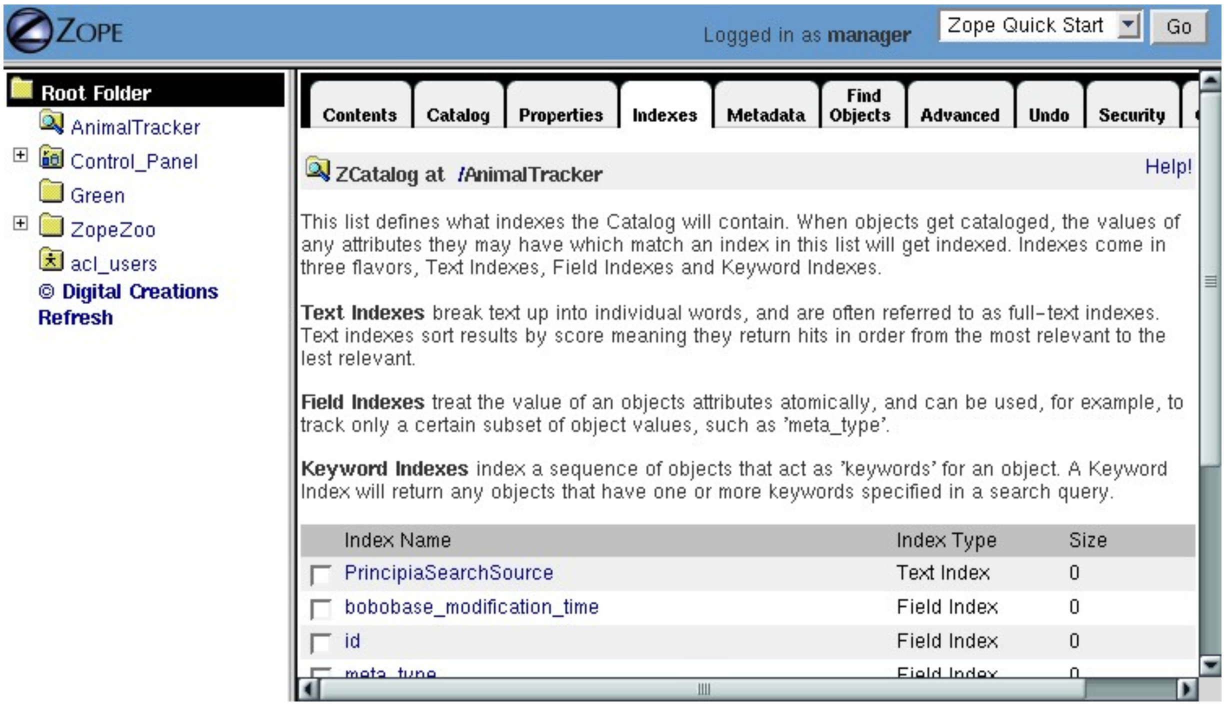
When you search a ZCatalog you are not searching through your objects one by one. That would take far too much time if you had a lot of objects. Before you search a ZCatalog, it looks at your objects and remembers whatever you tell it to remember about them. This process is called *indexing*. From then on, you can search for certain criteria and the ZCatalog will return objects that match the criteria you provide.

A good way to think of an index in a ZCatalog is just like an index in a book. For example, in a book's index you can look up the word *Python*:

```
Python: 23, 67, 227
```

The word *Python* appears on three pages. Zope indexes work like this except that they map the search term, in this case the word *Python*, to a list of all the objects that contain it, instead of a list of pages in a book.

In Zope 2.4, indexes can be added and removed from a Catalog using a new, "pluggable" index interface as shown in Figure 9−3:



ZCatalog at /AnimalTracker

This list defines what indexes the Catalog will contain. When objects get cataloged, the values of any attributes they may have which match an index in this list will get indexed. Indexes come in three flavors, Text Indexes, Field Indexes and Keyword Indexes.

Text Indexes break text up into individual words, and are often referred to as full-text indexes. Text indexes sort results by score meaning they return hits in order from the most relevant to the lest relevant.

Field Indexes treat the value of an objects attributes atomically, and can be used, for example, to track only a certain subset of object values, such as 'meta_type'.

Keyword Indexes index a sequence of objects that act as 'keywords' for an object. A Keyword Index will return any objects that have one or more keywords specified in a search query.

Index Name	Index Type	Size
PrincipiaSearchSource	Text Index	0
bobobase_modification_time	Field Index	0
id	Field Index	0
meta_type	Field Index	0

Figure 9−3 Managing indexes

Here, you can see that ZCatalogs come with some predefined indexes. Each index has a name, like *PrincipiaSearchSource*, and a type, like *TextIndex*.

When you catalog an object the Catalog uses each index to examine the object. The catalog consults attributes and methods to find an object's value for each index. For example, in the case of the DTML Documents cataloged with a *PrincipiaSearchSource* index, the Catalog calls each document's *PrincipiaSearchSource* method and records the results in its *PrincipiaSearchSource* index. If the Catalog cannot find an attribute or method for an index, then it ignores it. In other words it's fine if an object does not support a given index. There are four kinds of indexes:

TextIndex
: Searches text. Use this kind of index when you want a full−text search.

FieldIndex
: Searches objects for specific values. Use this kind of index when you want to search date objects, numbers, or specific strings.

KeywordIndex
: Searches collections of specific values. This index is like a FieldIndex, but it allows you to search collections rather than single values.

PathIndex
: Searches for all objects that contain certain URL path elements. For example, you could search for all the objects whose paths begin with `/Animals/Zoo`.

We'll examine these different indexes more closely later in the chapter. New indexes can be created from the *Indexes* view of a ZCatalog. There, you can enter the *name* and select a *type* for your new index. This creates a new empty index in the ZCatalog. To populate this index with information, you need to Go to the *Advanced* view and click the the *Update Catalog* button. Recataloging your content may take a while if you have lots of cataloged objects.

To remove an index from a Catalog, select the Indexes and click on the *Delete* button. This will delete the index and all of its indexed content. As usual, this operation is undoable.

Defining Meta Data

The ZCatalog can not only index information about your object, but it can also store information about your object in a *tabular database* called the *Meta−Data Table*. The *Meta−Data Table* works similarly to a relational database table, it consists of one or more *columns* that define the *schema* of the table. The table is filled with *rows* of information about cataloged objects. These rows can contain information about cataloged objects that you want to store in the table. Your meta data columns don't need to match your Catalog's indexes. Indexes allow you to search; meta−data allows you to report search results.

The Meta−Data Table is useful for generating search reports. It keeps track of information about objects that goes on your report forms. For example, if you create a Meta−Data Table column called *absolute_url*, then your report forms can use this information to create links to your objects that are returned in search results.

To add a new Meta−Data Table column, type in the name of the column on the *Meta−Data Table* view and click *Add*. To remove a column from the Meta−Data Table, select the column check box and click on the *Delete* button. This will delete the column and all of its content for each row. As usual, this operation is undoable. Next let's look more closely at how to search a Catalog.

Searching Catalogs

You can search a Catalog by passing it search terms. These search terms describe what you are looking for in one or more indexes. The Catalog can glean this information from the web request, or you can pass this information explicitly from DTML or Python. In response to a search request, a Catalog will return a list of records corresponding to the cataloged objects that match the search terms.

Searching with Forms

In this chapter you used the *Z Search Interface* to automatically build a Form/Action pair to query a Catalog (the Form/Action pattern is discussed in Chapter 4, "Dynamic Content with DTML"). The *Z Search Interface* builds a very simple form and a very simple report. These two methods are a good place to start understanding how Catalogs are queried and how you can customize and extend your search interface.

Suppose you have a catalog that holds news items. Each news item has contents, an author and a date. Your catalog has three indexes that correspond to these attributes. The contents index is a text index, and the author and date indexes are field indexes. Here is the search form that would allow you to query such a catalog:

```
<dtml-var standard_html_header>

<form action="Report" method="get">
<h2><dtml-var document_title></h2>
Enter query parameters:<br><table>

<tr><th>Content</th>
    <td><input name="content" width=30 value=""></td></tr>
<tr><th>Author</th>
    <td><input name="author" width=30 value=""></td></tr>
<tr><th>Date</th>
    <td><input name="date"  width=30 value=""></td></tr>

<tr><td colspan=2 align=center>
<input type="SUBMIT" value="Submit Query">
</td></tr>
</table>
</form>

<dtml-var standard_html_footer>
```

This form consists of three input boxes named *content*, *author*, and *date*. These names of the input form elements match the names of the indexes in the catalog. These names must match the names of the catalog's indexes for the catalog to find the search terms. Here is a report form that works with the search form:

```
<dtml-var standard_html_header>

<table>
  <dtml-in NewsCatalog>
  <tr>
    <td><dtml-var author></td>
    <td><dtml-var date></td>
  </tr>
  </dtml-in>
</table>

<dtml-var standard_html_footer>
```

There are a few things going on here which merit closer examination. The heart of the whole thing is the *in* tag.:

```
        <dtml-in NewsCatalog>
```

This tag calls the *NewsCatalog* Catalog. Notice how the form parameters from the search form (content, author, date) are not mentioned here at all. Zope automatically makes sure that the query parameters from the search form are given to the Catalog. All you have to do is make sure the report form calls the Catalog. Zope locates the search terms in the web request and passes them to the Catalog.

The Catalog returns a sequence of *Record Objects* (just like ZSQL Methods). These record objects correspond to *search hits,* which are objects that match the search criteria you typed in. For a record to match a search, it must match all criteria for each specified index. So if you enter an author and some search terms for the contents, the Catalog will only return records that match both the author and the contents.

Record objects had an attribute for every column in the database table. Record objects for Catalogs work very similarly, except that a Catalog Record object has an attribute for every column in the Meta−Data Table. In fact, the purpose of the Meta−Data Table is to define the schema for the Record objects that Catalog queries return.

Searching from Python

DTML makes querying a Catalog from a form very simple. For the most part, DTML will automatically make sure your search parameters are passed properly to the Catalog.

Sometimes though you may not want to search a Catalog from a web form; some other part of your application may want to query a Catalog. For example, suppose you want to add a sidebar to the Zope Zoo that shows news items that only relate to the animals in the section of the site that you are currently looking at. As you've seen, the Zope Zoo site is built up from Folders that organize all the sections according to animal. Each Folder's id is a name that specifies the group or animal the folder contains. Suppose you want your sidebar to show you all the news items that contain the id of the current section. Here is a Script called *relevantSectionNews* that queries the news Catalog with the currentfolder's id:

```
        ## Script (Python) "relevantSectionNews"
        ##
        """ Returns news relevant to the current folder's id """
        id=context.getId()
        return context.NewsCatalog({'content' : id})
```

This script queries the NewsCatalog by calling it like a method. Catalog's expect a *mapping* as the first argument when they are called. The argument maps the name of an index to the search terms you are looking for. In this case, the *content* index will be queried for all news items that contain the name of the current Folder. To use this in your sidebar, just edit the Zope Zoo's standard_html_header to use the relevantSectionNews script:

```
        <html>
        <body>
        <dtml−var style_sheet>
        <dtml−var navigation>
        <ul>
        <dtml−in relevantSectionNews>
          <li><a href="&dtml−absolute_url;"><dtml−var title></a></li>
        </dtml−in>
        </ul>
```

This method assumes that you have defined *absolute_url* and *title* as meta−data columns in the news Catalog. Now, when you are in a particular section, the sidebar will show a simple list of links to news items that contain the id of the current animal section you are viewing.

Searching and Indexing Details

Earlier you saw that the Catalog supports three types of indexes, text indexes, field indexes and keyword indexes. Let's examine these indexes more closely to understand what they are good for and how to search them.

Searching Text Indexes

A Text Index is used to index text. After indexing, you can search the index for objects that contain certain words. Text Indexes support a rich search grammar for doing more advanced searches than just looking for a word. ZCatalog's Text Index can:

- Search for Boolean expressions like "word1 AND word2". This will search for all objects that contain *both* "word1" and "word2". Valid Boolean operators include AND, OR, and AND NOT.
- Control search order with parenthetical expressions "(word1 AND word2) OR word3)". This will return objects containing "word1" and "word2" *or* just objects that contain the term "word3".
- If you use a special kind of *Vocabulary* object (explained a little further on) you can search using simple wild cards like "Z*", which returns all words that begin with "Z".

All of these advanced features can be mixed together. For example, "((bob AND uncle) AND NOT Zoo*)" will return all objects that contain the terms "bob" and "uncle" but will not include any objects that contain words that start with "Zoo" like "Zoologist", "Zoology", or "Zoo" itself.

Querying a TextIndex with these advanced features works just like querying it with the original simple features. In the HTML search form for DTML Documents, for example, you could enter "Koala AND Lion" and get all documents about Koalas and Lions. Querying a *TextIndex* from Python with advanced features works much the same; suppose you want to change your *relevantSectionNews* Script to not include any news items that contain the word "catastrophic":

```
## Script (Python) "relevantSectionNews"
##
""" Returns relevant, non−catastropic news """"
id=context.getId()
return context.NewsCatalog(
         {'content' : id + ' AND NOT catastrophic'}
        )
```

TextIndexes are very powerful. When mixed with the Automatic Cataloging pattern described later in the chapter, they give you the ability to automatically free−text search all of your objects as you create and edit them.

Vocabularies

Vocabularies are used by text indexes. A vocabulary is an object that manages language specific text indexing options. In order for the ZCatalog to work with any kind of language, it must understand certain behaviors of that language. For example, all languages:

- have a different concept of *words*. In English and many other languages, words are defined by white space boundaries, but in other languages, like Chinese and Japanese, words are defined by their contextual usage.
- have different concepts of *stop words*. A stop word is a common word that should be ignored by indexes. The French word *nous* would be extremely common in French text and should probably be removed as a stop word, but in English text it might make perfect sense to catalog this word because it

is very infrequent.
- have different concepts of synonymous, The synonym pair *automobile/car* would not make sense in any language but English.
- have different concepts of stemming. In English, it is common for text indexers to strip suffixes like *ing* from words, so that *bake* and *baking* match the same word. This is called stemming. These suffix strippings would only make sense to English, and other languages would want to provide their own stemming (or none at all).

Current Vocabularies

There are a number of vocabularies currently available for ZCatalog:

Plain Vocabularies
: Plain vocabularies are very simple and do minimal English language specific tasks.

Globbing Vocabularies
: Globbing vocabularies are more complex vocabularies that allow wild card searches on English text to be performed. The down side of them is that they consume a lot more memory and database space than plain vocabularies.

The idea behind Vocabularies is to customize the way text in any language is indexed. Because of this, other languages may be supported in the future by people who create a Vocabulary specific to their language. Creating your own Vocabulary is an advanced topic, and beyond the scope of this book.

Using Vocabularies

When you create a new ZCatalog, the ZCatalog add form has a select box for you to choose a vocabulary to use. If you do not select a vocabulary, the ZCatalog automatically creates a Plain Vocabulary for you, and adds it to the ZCatalog's contents (this can be seen on the *Contents* view of the AnimalTracker you created for the examples in this chapter).

To use a Globbing Vocabulary or any other kind of Vocabulary, you must create it first before you create the Catalog you want to use it on. A ZCatalog can use any Vocabulary inside its contents or any Vocabulary that it can find *above* it in the Zope Folder hierarchy.

Searching Field Indexes

FieldIndexes differ slightly from TextIndexes. A TextIndex will treat the value it finds in your object, for example the contents of a News Item, like text. This means that it breaks the text up into words and indexes all the individual words.

A FieldIndex does not break up the value it finds. Instead, it indexes the entire value it finds. This is very useful for tracking objects that have traits with fixed values.

In the news item example, you created two FieldIndexes, *date* and *author*. With the existing search form, these fields are not very useful. To use them more effectively you have to customize your search form a little. Before doing that though, let's consider some use cases for these indexes.

The *date* index lets you search for News Items by the time they were created. The existing search form is not very useful though because you have to type in *exactly* the time you were looking for, right down to the second, in the text box to get any hits. This is obviously not very useful. It would be better to search for a *range* of dates, like all of the News Items added in the last 24 hours, or all of the next Items from last month.

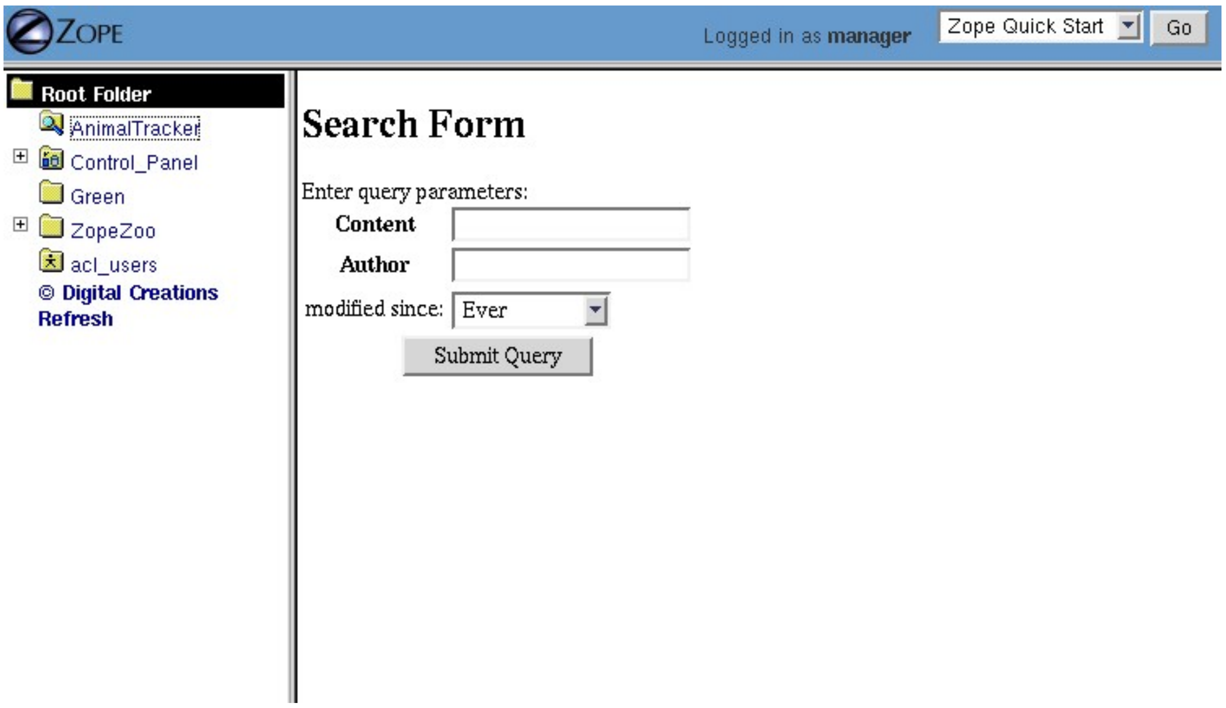
The *author* index lets you search for News Items by certain authors. Unless you know exactly the name of the author you are looking for though, you will not get any results. It would be better to be able to select from a list of all the *unique* authors indexed by the author index.

FieldIndexes are designed to do both range searching and searching for a unique value in the index. To take advantage of these features, you need only change your search form a little bit. Let's try the first example, range searching with dates.

Like TextIndexes, FieldIndexes can be passed special options to enable these features. These special features need to be passed in as form elements that get turned into Catalog queries. Here is the search form used in the previous section *Searching with Forms*, but with some new form elements added to enable searching for News Items modified since "Yesterday", "Last Week", "Last Month", "Last Year" or "Ever":

```
<dtml-var standard_html_header>

<form action="Report" method="get">
<h2><dtml-var document_title></h2>
Search for News Items:<br><table>

<tr><th>Content</th>
    <td><input name="content" width=30 value=""></td></tr>
<tr><th>Author</th>
    <td><input name="author" width=30 value=""></td></tr>
<tr>
  <td><p>modified since:</p></td>
  <td>
    <input type="hidden" name="date_usage" value="range:min">
    <select name="date:date">
      <option value="<dtml-var expr="ZopeTime(0)" >">Ever</option>
      <option value="<dtml-var expr="ZopeTime() - 1" >">Yesterday</option>
      <option value="<dtml-var expr="ZopeTime() - 7" >">Last Week</option>
      <option value="<dtml-var expr="ZopeTime() - 30" >">Last Month</option>
      <option value="<dtml-var expr="ZopeTime() - 365" >">Last Year</option>
    </select>
  </td>
</tr>

<tr><td colspan=2 align=center>
<input type="SUBMIT" value="Submit Query">
</td></tr>
</table>
</form>
<dtml-var standard_html_footer>
```

This should make your search form look like Figure 9−4.



Search Form

Enter query parameters:

Content	
Author	
modified since:	Ever

Submit Query

Figure 9−4 Range searching by Date

This HTML form changes the *date* format from the old search form. Instead of just a text box, it offers you a selection box where you can choose a date. But remember, this is a *range* search. Can you spot the part that tells the *date* FieldIndex to search by range? Here it is:

```
<input type="hidden" name="date_usage" value="range:min">
```

This is a special kind of HTML form element called a *hidden* element. It does not show up anywhere on the search form that you look at, but it is still passed into Zope when you submit the form. This special element, called *date_usage* tells the *date* FieldIndex that the value in the *date* form element is a *minimum range boundary*. This means that the FieldIndex will not just return objects that have that date, but it will return objects that have that date or *any later date*.

Any kind of FieldIndex can be told what kind of range specifiers to use by adding an additional search argument that suffixes the index name with "_usage". In addition to specifying a minimum range boundary, you specify a *maximum* range boundary by changing the hidden form element to:

```
<input type="hidden" name="date_usage" value="range:max">
```

This will cause the search form to return all News Items modified *before* the specified date, instead of after.

The "_usage" syntax can also be used when calling a Catalog directly from a script, like this Script, *relevantRecentSectionNews*:

```
## Script (Python) "relevantRecentSectionNews"
##
""" Return relevant, and recent, news for this section """
id=context.getId()
return context.NewsCatalog(
         {'content'    : id,
          'date'       : ZopeTime() - 7,
          'date_usage' : 'range:min',
         }
```

```
                )
```

This works just like your old *relevantSectionNews* script, except that it only shows news items created in the last week.

You can also supply both a minimum and maximum range boundary. There's one catch to this, however. Normally if you specify no range boundary or just one boundary, ZCatalog uses the value you pass in as the search term. But when you provide *two* range boundaries, the ZCatalog needs two values, not one. Here is the *relevantRecentSectionNews* Script above with some slight modification to provide a *list* of date objects instead of just one:

```
## Script (Python) "relevantRecentSectionNews"
##
"""
Return relevant news modified in the last month, but not the
last week
"""
id=context.getId()
return context.NewsCatalog(
         {'content'    : id,
          'date'       : [ZopeTime() - 30, ZopeTime() - 7],
          'date_usage' : 'range:min:max',
         }
        )
```

This script will return all of the relevant News Items modified in the last month, but *not* in the last week. When using two range specifiers, it is important to make sure you get the order of the values to correctly match the order of the range specifiers. If you were to accidentally switch the "min" and "max" around, but didn't switch around the two dates, then you will get *no* search results because you are making a query that doesn't make sense (providing a minimum value that is larger than the maximum value).

The second use case you considered above was being able to search from a list of all unique authors. There is a special method on the ZCatalog that does exactly this called *uniqueValuesFor*. The uniqueValuesFor method returns a list of unique values for a certain index. Let's change your search form yet again, and replace the original *author* input box with something a little more useful:

```
<dtml-var standard_html_header>

<form action="Report" method="get">
<h2><dtml-var document_title></h2>
Search for News Items:<br><table>

<tr><th>Content:</th>
    <td><input name="content" width=30 value=""></td></tr>
<tr valign="top">
   <td><p>Author:</p></td>

   <td>
     <select name="author:list" size=6 MULTIPLE>
     <dtml-in expr="AnimalTracker.uniqueValuesFor('author')">
       <option value="<dtml-var sequence-item>">
       <dtml-var sequence-item></option>
     </dtml-in>
     </select>
   </td>
 </tr>

<tr>
  <td><p>modified since:</p></td>
  <td>
```

```
        <input type="hidden" name="date_usage" value="range:min">
        <select name="date:date">
          <option value="<dtml-var "ZopeTime(0)" >">Ever</option>
          <option value="<dtml-var "ZopeTime() - 1" >">Yesterday</option>
          <option value="<dtml-var "ZopeTime() - 7" >">Last Week</option>
          <option value="<dtml-var "ZopeTime() - 30" >">Last Month</option>
          <option value="<dtml-var "ZopeTime() - 365" >">Last Year</option>
        </select>
      </td>
    </tr>

    <tr><td colspan=2 align=center>
    <input type="SUBMIT" name="SUBMIT" value="Submit Query">
    </td></tr>
    </table>
    </form>
    <dtml-var standard_html_footer>
```

The new, important bit of code added to the search form is:

```
    <select name="author:list" size=6 MULTIPLE>
    <dtml-in expr="AnimalTracker.uniqueValuesFor('author')">
      <option value="<dtml-var sequence-item>">
      <dtml-var sequence-item></option>
    </dtml-in>
    </select>
```

The HTML was also changed a bit to make the on−screen presentation make sense.

In this example, you are changing the form element *author* from just a simple text box to an HTML multiple select box. This box contains a unique list of all the authors that are indexed in the *author* FieldIndex. Now, your search form should look like Figure 9−5.

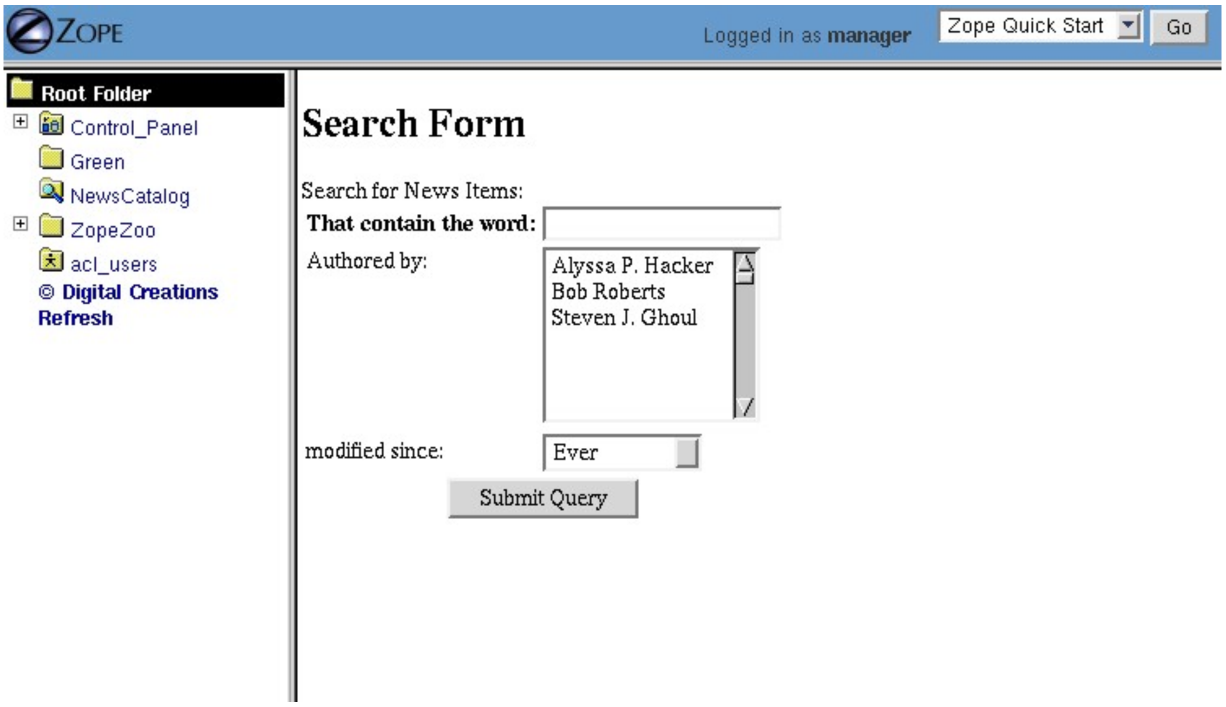


Figure 9−5 Range searching and unique Authors

That's it. You can continue to extend this search form using HTML form elements to be as complex as you'd like. In the next section, we'll show you how to use the next kind of index, keyword indexes.

Searching Keyword Indexes

A *KeywordIndex* indexes a sequence of keywords for objects and can be queried for any objects that have one or more of those keywords.

Suppose that you have a number of Image objects that have a *topics* property. The *topics* property is a lines property that lists the relevant topics for a given Image, for example, "Portraits", "19th Century", and "Women" for a picture of Queen Victoria.

The topics provide a way of categorizing Images. Each Image can belong in one or more categories depending on its *topics* property. For example, the portrait of Queen Victoria belongs to three categories and can thus be found by searching for any of the three terms.

You can use a *KeyWord* index to search the *topics* property. Define a *KeyWord* index with the name *topics* on your ZCatalog. Then catalog your Images. Now you should be able to find all the Images that are portraits by creating a search form and searching for "Portraits" in the *topics* field. You can also find all pictures that represent 19th Century subjects by searching for "19th Century".

It's important to realize that the same Image can be in more than one category. This gives you much more flexibility in searching and categorizing your objects than you get with a field index. Using a field index your portrait of Queen Victoria can only be categorized one way. Using a keyword index it can be categorized a couple different ways.

Often you will use a small list of terms with *KeyWord* indexes. In this case you may want to use the *uniqueValuesFor* method to create a custom search form. For example here's a snippet of DTML that will create a multiple select box for all the values in the *topics* index:

```
<select name="topics:list" multiple>
<dtml-in expr="uniqueValuesFor('topics')">
  <option value="&dtml-sequence-item;"><dtml-var sequence-item></option>
</dtml-in>
</select>
```

Using this search form you can provide users with a range of valid search terms. You can select as many topics as you want and Zope will find all the Images that match one or more of your selected topics. Not only can each object have several indexed terms, but you can provide several search terms and find all objects that have one or more of those values.

Searching Path Indexes

Path indexes allow you to search for objects based on their location in Zope. Suppose you have an object whose path is `/zoo/animals/Africa/tiger.doc`. You can find this object with the path queries: `/zoo`, or `/zoo/animals`, or `/zoo/animals/Africa`. In other words, a path index allows you to find objects within a given folder (and below).

If you place related objects within the same folders, you can use path indexes to quickly located these objects. For example:

```
<h2>Lizard Pictures</h2>
```

```
<p>
<dtml-in expr="Catalog(meta_type='Image',
                       path='/Zoo/Animals/Lizard')">
<a href="&dtml-absolute_url;"><dtml-var title></a>
</dtml-in>
</p>
```

This query searches a catalog for all images that are located within the `/Zoo/Animals/Lizard` folder and below. It creates a link to each image.

Depending on how you choose to arrange objects in your site, you may find that a path indexes are more or less effective. If you locate objects without regard to their subject (for example, if objects are mostly located in user "home" folders) then path indexes may be of limited value. In these cases, key word and field indexes will be more useful.

Advanced Searching with Records

A new feature in Zope 2.4 is the ability to query indexes more precisely using record objects. Record objects contain information about how to query an index. Records are Python objects with attributes, or mappings. Different indexes support different record attributes.

Keyword Index Record Attributes

`query`
: Either a sequence of words or a single word. (mandatory)

`operator`
: Specifies whether all keywords or only one need to match. Allowed values: `and`, `or`. (optional, default: 'or')

For example:

```
# big or shiny
results=Catalog(categories=['big, 'shiny'])

# big and shiny
results=Catalog(categories={'query':['big','shiny'],
                                     'operator':'and'})
```

The second query matches objects that have both the keywords "big" and "shiny". Without using the record syntax you can only match objects that are big or shiny.

Field Index Record Attributes

`query`
: Either a sequence of objects or a single value to be passed as query to the index (mandatory)

`range`
: Defines a range search on a Field Index (optional, default: not set).
: **Allowed values:**

    `min`
    : Searches for all objects with values larger than the minimum of the values passed in the `query` parameter.

    `max`

Searches for all objects with values smaller than the maximum of the values passed in the `query` parameter.

`minmax`
: Searches for all objects with values smaller than the maximum of the values passed in the `query` parameter and larger than the minimum of the values passwd in the `query` parameter.

For example:

```
# items modified in the last week
results=Catalog(bobobase_modification_time={
                  'query':DateTime() - 7,
                  'range': 'min'}
                )
```

This query matches objects with a `bobobase_modification_time` of less than `DateTime() -7`. Compare this query with one defined in `relevantRecentSectionNews` earlier in this chapter which uses `date_usage` to accomplish the same query.

Text Index Record Attributes

`query`
: Either a sequence of words (seperated by white space) or a single word to be passed as query to the index. (mandatory)

`operator`
: Specifies how to combine the search terms. (optional, default: 'or').

Allowed values:

`and`
: All terms must be present.

`or`
: At least one term must be present.

`andnot`
: The first term must be present, but none of the rest of the terms.

There's not much reason to use record queries with text indexes since you can embed the operator information in the query string itself in a very flexible manner.

Path Index Record Attributes

`query`
: Path to search for either as a string (e.g. "/Zoo/Birds") or list (e.g. ["Zoo", "Birds"]). (mandatory)

`level`
: The path level to begin searching at. (optional, default: '0')

Suppose you have a collection of objects with these paths:

1. /aa/bb/aa
2. /aa/bb/bb
3. /aa/bb/cc
4. /bb/bb/aa
5. /bb/bb/bb
6. /bb/bb/cc

7. /cc/bb/aa
8. /cc/bb/bb
9. /cc/bb/cc

Here are some examples queries and their results to show how the `level` attribute works:

- query='/aa/bb', level=0 returns 1, 2, 3
- query='/bb/bb', level=0 returns 4, 5, 6
- query='/bb/bb', level=1 returns 2, 5, 8
- query='/bb/bb', level=−1 returns 2, 4, 5, 6, 8
- query='/xx', level=−1 returns none

You can use the level attribute to flexibly search different parts of the path.

As of Zope 2.4.1, you can also include level information in a search without using a record. Simply use a tuple containing the query and the level. Here's an example tuple: `("/aa/bb", 1)`.

Creating Records in HTML

You can also perform record queries using HTML forms. Here's an example showing how to create a search form using records:

```
<form action="Report" method="get">
<table>
<tr><th>Search Terms (must match all terms)</th>
    <td><input name="content.query:record" width=30 value=""></td></tr>
    <input type="hidden" name="content.operator:record" value="and">
<tr><td colspan=2 align=center>
<input type="SUBMIT" value="Submit Query">
</td></tr>
</table>
</form>
```

For more information on creating records in HTML see the section "Passing Parameters to Scripts" in Chapter 10, Advanced Zope Scripting.

Stored Queries

While the main use of the Catalog is to provide interactive searching, you can also use *stored queries* to categorize and organize your site. For example, in the section on keyword indexes you saw how you can use the Catalog and properties to search for categories of Images such as portraits. In addition to providing interactive searching for categories of Images you can create web pages with canned queries. So for example, here's some DTML that you could use for a page that displays all your portraits:

```
<dtml-var standard_html_header>

<h1>Portraits</h1>

<dtml-in expr="ImageCatalog({'topics':'Portraits'})">
<p>
<dtml-var sequence-item>
<dtml-var title_or_id>
</p>
</dtml-in>

<dtml-var standard_html_footer>
```

The dynamic nature of this page is not visible to the viewer. However, just add another portrait, update the catalog and this page will automatically include the new Image.

This technique can be very powerful. Not only can you organize and display public resources, but you can easily institute workflow systems by tagging objects with properties to indicate their state and cataloging them. After that it's easy for you to create pages for different people that show which objects need their attention. This technique is even more powerful when using the *Automatic Cataloging* pattern.

Automatic Cataloging

Automatic Cataloging is an advanced Catalog usage pattern that keeps objects up to date as they are changed. It requires that as objects are created, changed, and destroyed, they are automatically tracked by a ZCatalog. This usually involves the objects notifying the Catalog when they are created, changed, or deleted.

This usage pattern has a number of advantages in comparison to mass cataloging. Mass cataloging is simple but has drawbacks. The total amount of content you can index in one transaction is equivalent to the amount of free virtual memory available to the Zope process, plus the amount of temporary storage the system has. In other words, the more content you want to index all at once, the better your computer hardware has to be. Mass cataloging works well for indexing up to a few thousand objects, but beyond that automatic indexing works much better.

Another major advantage of automatic cataloging is that it can handle objects that change. As objects evolve and change, the index information is always current, even for rapidly changing information sources like message boards.

In this section, we'll show you an example that creates "news" items thatpeople can add to your site. These items will get automatically cataloged. This example consists of two steps:

- Creating a new type of object to catalog.
- Creating a Catalog to catalog the newly created objects.

As of Zope 2.3, none of the "out−of−the−box" Zope objects support automatic cataloging. This is for backwards compatibility reasons. For now, you have to define your own kind of objects that can be cataloged automatically. One of the ways this can be done is by defining a *ZClass*.

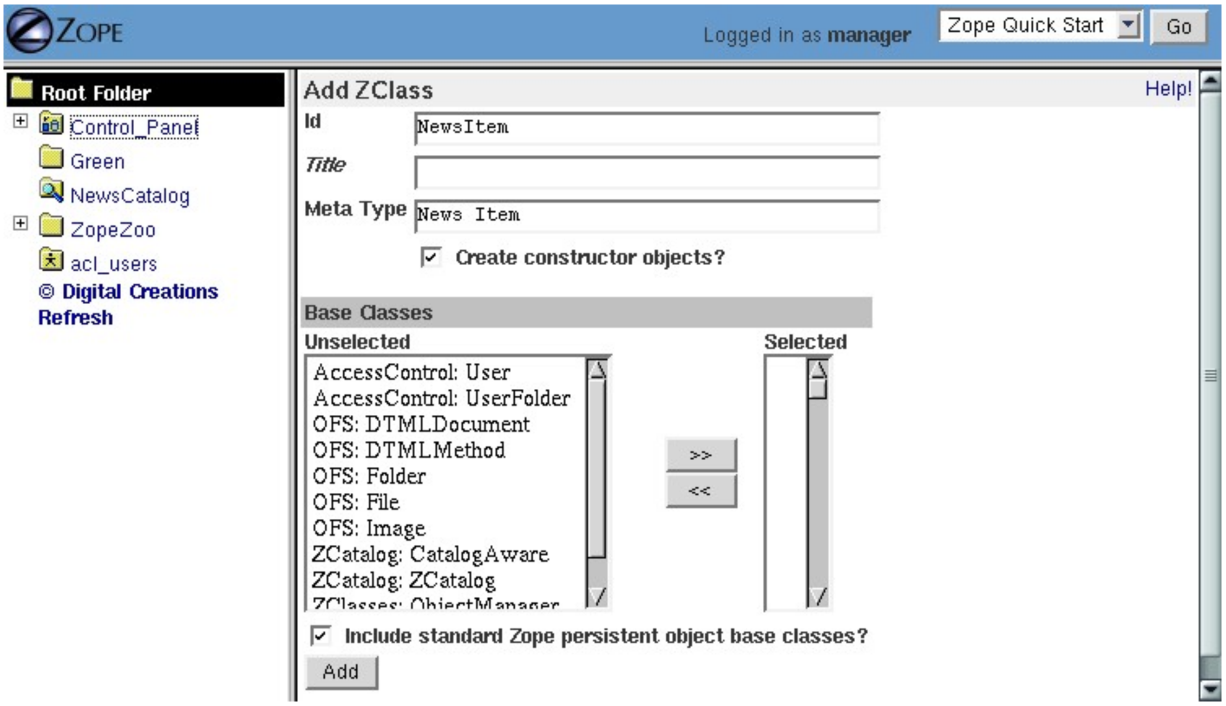
A ZClass is a Zope object that defines new types of Zope objects. In a way, a ZClass is like a blueprint that describes how new Zope objects are built. Consider a news item as discussed in examples earlier in the chapter. News items not only have content, but they also have specific properties that make them news items. Often these Items come in collections that have their own properties. You want to build a News site that collects News Items, reviews them, and posts them online to a web site where readers can read them.

In this kind of system, you may want to create a new type of object called a *News Item*. This way, when you want to add a new news item to your site, you just select it from the product add list. If you design this object to be automatically cataloged, then you can search your news content very powerfully. In this example, you will just skim a little over ZClasses, which are described in much more detail in Chapter 14, "Extending Zope."

New types of objects are defined in the *Products* section of the Control Panel. This is reached by clicking on the Control Panel and then clicking on *Product Management*. Products contain new kinds of ZClasses. On this screen, click "Add" to add a New product. You will be taken to the Add form for new Products.

Name the new Product "News" and click "Generate". This will take you back to the Products Management view and you will see your new Product.

Select the *News* Product by clicking on it. This new Product looks a lot like a Folder. It contains one object called *Help* and has an Add menu, as well as the usual Folder "tabs" across the top. To add a new ZClass, pull down the Add menu and select *ZClass*. This will take you to the ZClass add form, as shown in Figure 9−6.



Add ZClass

Id	NewsItem
Title	
Meta Type	News Item

☑ Create constructor objects?

Base Classes

Unselected

AccessControl: User
AccessControl: UserFolder
OFS: DTMLDocument
OFS: DTMLMethod
OFS: Folder
OFS: File
OFS: Image
ZCatalog: CatalogAware
ZCatalog: ZCatalog
ZClasses: ObjectManager

Selected

☑ Include standard Zope persistent object base classes?

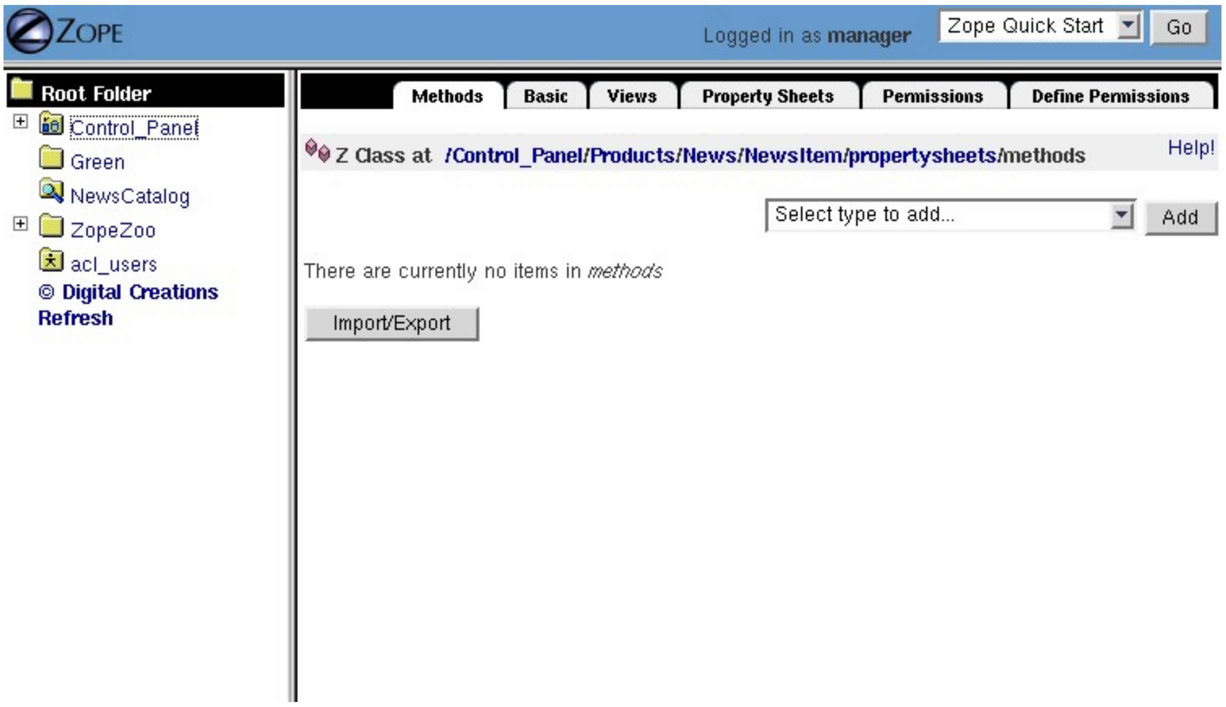
Figure 9−6 ZClass add form

This is a complicated form which will be explained in much more detail in Chapter 14, "Extending Zope". For now, you only need to do three things to create your ZClass:

- Specify the Id "NewsItem" This is the name of the new ZClass.
- Specify the meta_type "News Item". This will be used to create the Add menu entry for your new type of object.
- Select *ZCatalog:CatalogAware* from the left hand *Base Classes* box, and click the button with the arrow pointing to the right hand *Base Classes* box. This should cause *ZCatalog:CatalogAware* to show up in the right hand window.

When you're done, don't change any of the other settings in the Form. To create your new ZClass, click *Add*. This will take you back to your *News* Product. Notice that there is now a new object called *NewsItem* as well as several other objects. The *NewsItem* object is your new ZClass. The other objects are "helpers" that you will examine more in Chapter 14, "Extending Zope".

Select the *NewsItem* ZClass object. Your view should now look like Figure 9−7.



Z Class at /Control_Panel/Products/News/NewsItem/propertysheets/methods

There are currently no items in *methods*

Figure 9−7 A ZClass Methods View

This is the *Methods* View of a ZClass. Here, you can add Zope objects that will act as *methods on your new type of object*. Here, for example, you can create DTML Methods or Scripts and these objects will become methods on any new *News Items* that are created. Before creating any methods however, let's review the needs of this new "News Item" object:

News Content
: The news Item contains news content, this is its primary purpose. This content should be any kind of plain text or marked up content like HTML or XML.

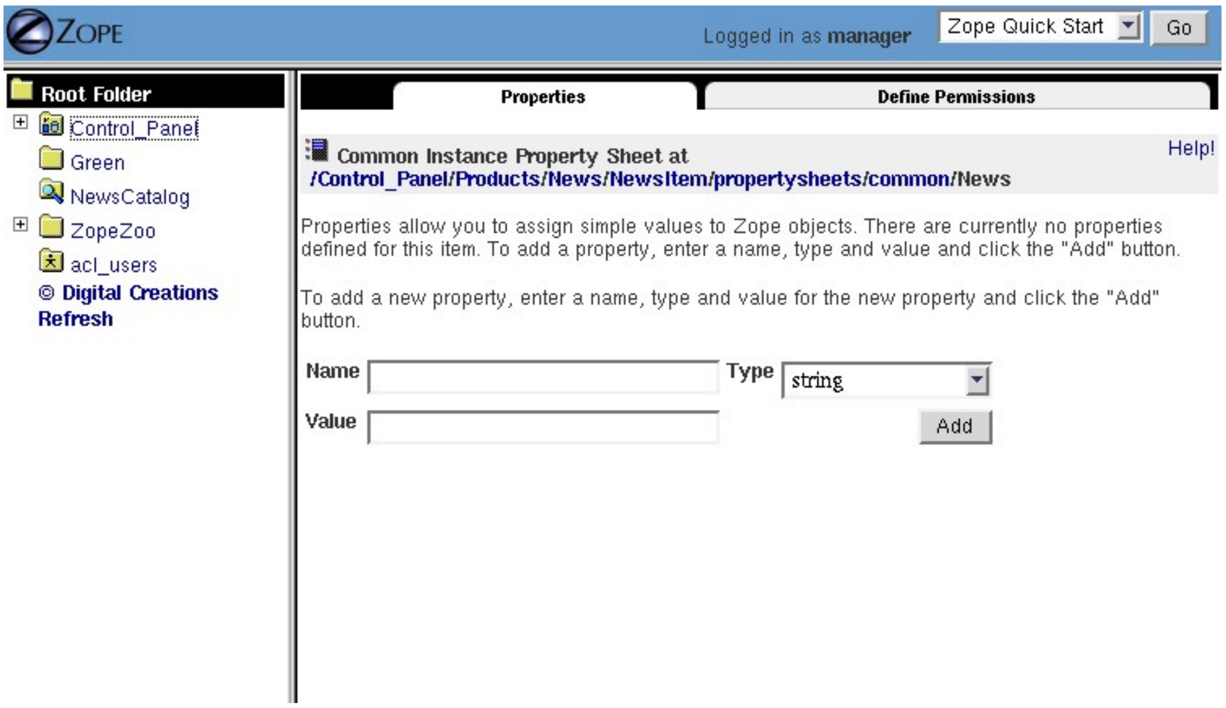
Author Credit
: The News Item should provide some kind of credit to the author or organization that created it.

Date
: News Items are timely, so the date that the item was created is important.

Keywords
: News Items fit into various lists of categories. By convention, these lists of categories are often called *keywords*.

You may want your new News Item object to have other properties, these are just suggestions. To add new properties to your News Item click on the *Property Sheets* tab. This takes you to the *Property Sheets* view.

Properties are added to new types of objects in groups called *Property Sheets*. Since your object has no property sheets defined, this view is empty. To add a New Property Sheet, click *Add Common Instance Property Sheet*, and give the sheet the name "News". Now click *Add*. This will add a new Property Sheet called *News* to your object. Clicking on the new Property Sheet will take you to the *Properties* view of the *News* Property Sheet, as shown in Figure 9−8.



Common Instance Property Sheet at /Control_Panel/Products/News/NewsItem/propertysheets/common/News

Properties allow you to assign simple values to Zope objects. There are currently no properties defined for this item. To add a property, enter a name, type and value and click the "Add" button.

To add a new property, enter a name, type and value for the new property and click the "Add" button.

Figure 9−8 The properties screen for a Property Sheet

This view is almost identical to the *Properties* view found on Folders and other objects. Here, you can create the properties of your News Item object. Create three new properties in this form:

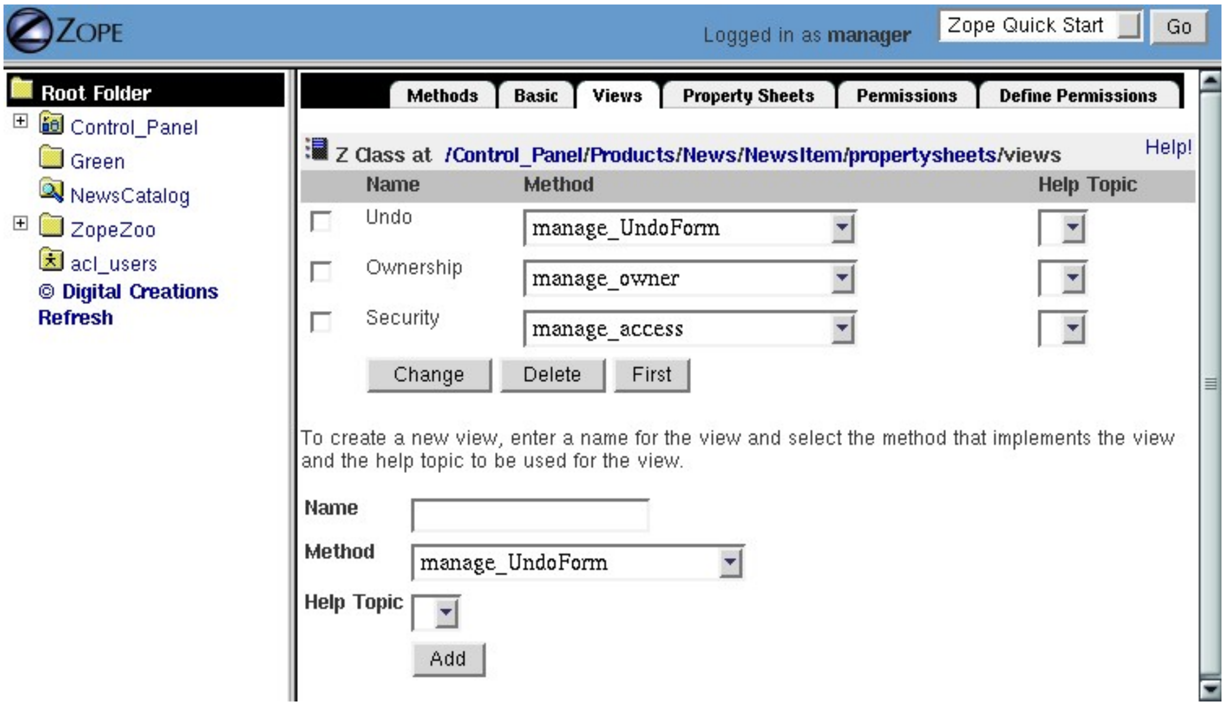
content
: This property's type should be *text*. Each newly created News Item will contain its own unique content property.

author
: This property's type should be *string*. This will contain the name of the news author.

date
: This property's type should be *date*. This will contain the time and date the news item was last updated. A *date* property requires a value, so for now you can enter the string "01/01/2000".

That's it! Now you have created a Property Sheet that describes your News Items and what kind of information they contain. Properties can be thought of as the *data* that an object contains. Now that we have the data all set, you need to create an *interface* to your new kind of objects. This is done by creating new *Views* for your object.

Click on the *Views* tab. This will take you to the *Views* view, as shown in Figure 9−9.



Z Class at /Control_Panel/Products/News/NewsItem/propertysheets/views

	Name	Method	Help Topic
☐	Undo	manage_UndoForm	
☐	Ownership	manage_owner	
☐	Security	manage_access	

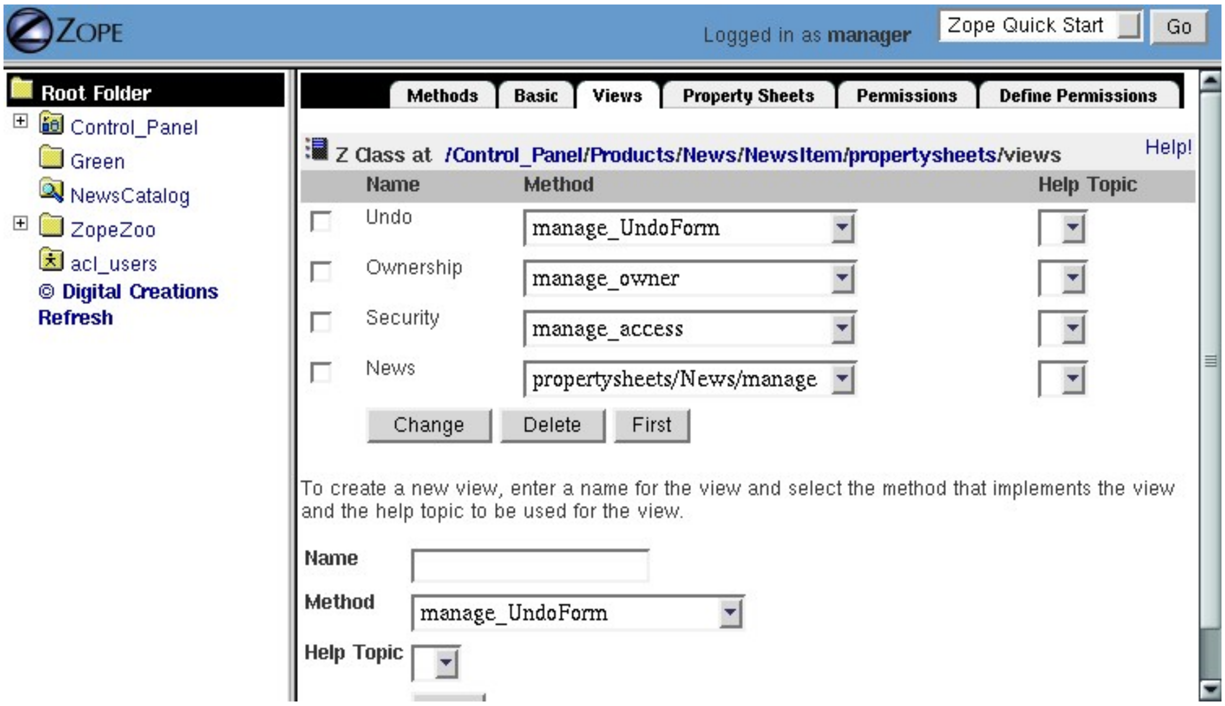
To create a new view, enter a name for the view and select the method that implements the view and the help topic to be used for the view.

Name	
Method	manage_UndoForm
Help Topic	

Figure 9−9 The Views view

Here, you can see that Zope has created three default Views for you. These views will be described in much more detail in Chapter 14, "Extending Zope", but for now, it suffices to say that these views define the tabs that your objects will eventually have.

To create a new view, use the form at the bottom of the Views view. Create a new View with the name "News" and select "propertysheets/News/manage" from the select box and click *Add*. This will create a new View on this screen under the original three Views, as shown in Figure 9−10.



Z Class at /Control_Panel/Products/News/NewsItem/propertysheets/views

	Name	Method	Help Topic
☐	Undo	manage_UndoForm	
☐	Ownership	manage_owner	
☐	Security	manage_access	
☐	News	propertysheets/News/manage	

To create a new view, enter a name for the view and select the method that implements the view and the help topic to be used for the view.

Name	
Method	manage_UndoForm
Help Topic	

Figure 9−10 The new News View

Since this View is going to give us the ability to edit the News Item, we want to make it the first view that you see when you select a News Item object. To change the order of the views, select the newly created *News* view and click the *First* button. This should move the new view from the bottom to the top of the list.

The final step in creating a ZClass is defining the methods for the class. Methods are defined on the *Methods* View. Click on the *Methods* tab and you will be taken to the Methods view. Select 'DTML Method' from the add list and add a new DTML Method with the id "index_html". This will be the default view of your news item. Add the following DTML to the new method:

```
<dtml-var standard_html_header>

<h1>News Flash</h1>

<p><dtml-var date></p>

<p><dtml-var author></p>

<P><dtml-var content></p>

<dtml-var standard_html_footer>
```

That's it! You've created your own kind of object called a *News Item*. When you go to the root folder, you will now see a new entry in your add list.

But don't add any new News Items yet, because the second step in this exercise is to create a Catalog that will catalog your new News Items. Go to the root folder and create a new catalog with the id *Catalog*.

Like the previous two examples of using a ZCatalog, you need to create Indexes and a Meta−Data Table that make sense for your objects. First, delete the default indexes in the new ZCatalog and create the following indexes to replace them:

content
: This should be a TextIndex. This will index the content of your News Items.

title
: This should be a TextIndex. This will index the title of your News Items.

author
: This should be a FieldIndex. This will index the author of the News Item.

date
: This should be a FieldIndex. This will index the date of the News Item.

After creating these Indexes, delete the default Meta−Data columns and add these columns to replace them:

- author
- date
- title
- absolute_url

After creating the Indexes and Meta−Data Table columns, create a search interface for the Catalog using the Z Search Interface tool described previously in this chapter.

Now you are ready to go. Start by adding some new News Items to your Zope. Go anywhere in Zope and select *News Item* from the add list. This will take you to the add Form for News items.

Give your new News Item the id "KoalaGivesBirth" and click *Add*. This will create a new News Item. Select the new News Item.

Notice how it has four tabs that match the four Views that were in the ZClass. The first View is *News*, this view corresponds to the *News* Property Sheet you created in the News Item ZClass.

Enter your news in the *contents* box:

```
Today, Bob the Koala bear gave birth to little baby Jimbo.
```

Enter your name in the *Author* box, and today's date in the *Date* box.

Click *Change* and your News Item should now contain some news. Because the News Item object is *CatalogAware*, it is automatically cataloged when it is changed or added. Verify this by looking at the *Cataloged Objects* tab of the ZCatalog you created for this example.

The News Item you added is the only object that is cataloged. As you add more News Items to your site, they will automatically get cataloged here. Add a few more items, and then experiment with searching the ZCatalog. For example, if you search for "Koala" you should get back the *KoalaGivesBirth* News Item.

At this point you may want to use some of the more advanced search forms that you created earlier in the chapter. You can see for example that as you add new News Items with new authors, the authors select list on the search form changes to include the new information.

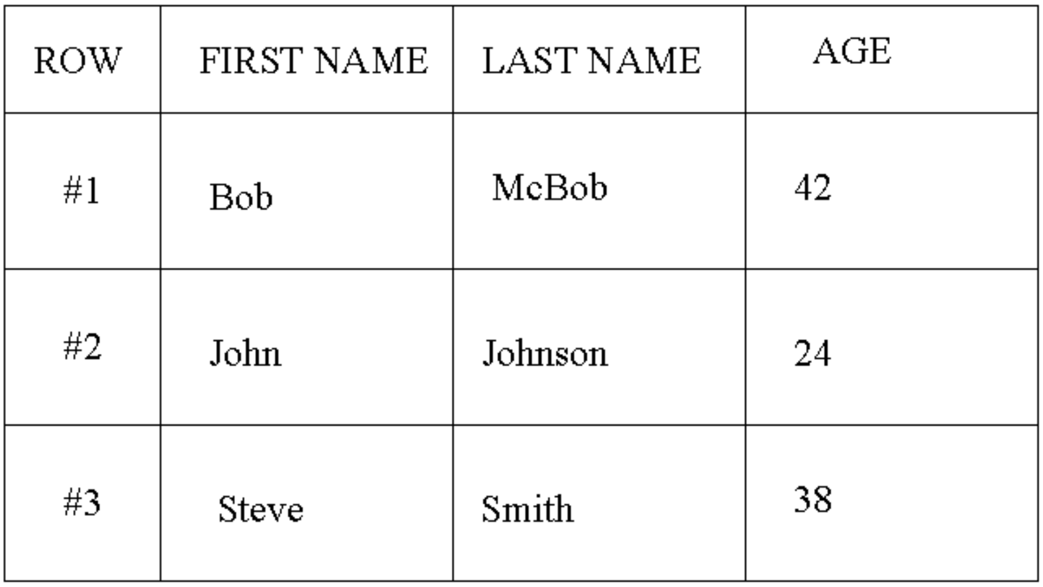
Conclusion

The cataloging features of ZCatalog allow you to search your objects for certain attributes very quickly. This can be very useful for sites with lots of content that many people need to be able to search in an efficient manner.

Searching the ZCatalog works a lot like searching a relational database, except that the searching is more object−oriented. Not all data models are object−oriented however, so in some cases you will want to use the ZCatalog, but in other cases you may want to use a relational database. The next chapter goes into more details about how Zope works with relational databases, and how you can use relational data as objects in Zope.

Chapter 12: Relational Database Connectivity

Zope uses an object database to store Zope objects. Relational databases such as Oracle, Sybase and PostgreSQL use a different store information in a different way. Relational databases store their information in tables as shown in Figure 10−1.



ROW	FIRST NAME	LAST NAME	AGE
#1	Bob	McBob	42
#2	John	Johnson	24
#3	Steve	Smith	38

Figure 10−1 Relational Database Table

Information in the table is stored in rows. The table's column layout is called the *schema*. A standard language, called the Structured Query Language (SQL) is used to query and change tables in relational databases.

Zope does not store its information this way. Zope's object database allows for many different types of objects that have many different types of relationships to each other. Relational data does not easily map onto objects since relational data assumes a much simpler table−oriented data model. Zope provides several mechanisms for taking relational data and using it in Zope's object−centric world including Database Adapters and SQL Methods which we will discuss in detail in this chapter.

The most common use for Zope's relational database support is to put existing relational databases on the web. For example, suppose your Human Resources Department has an employee database. Your database comes with tools to allow administrators run reports and change data. However, it is hard for employees to see their own records and perform simple maintenance such as updating their address when they move. By interfacing your relational database with Zope, your employees can use any web browser to view and update their records from the office or at home.

By using your relational data with Zope you get all of Zope's benefits including security, dynamic presentation, networking, and more. You can use Zope to dynamically tailor your data access, data presentation, and data management.

To use a relational database in Zope you must create two different Zope objects, a Database Connection and a Z SQL Method. Database Connections tell Zope how to connect to a relational database. Z SQL Methods describe an action to take on a database. Z SQL Methods use Database Connections to connect to relational databases. We'll look more closely at these two types of objects in this chapter.

Using Database Connections

Database Connections are used to establish and manage connections to external relational databases. Database Connections must be established before database methods can be defined. Moreover, every Z SQL Method must be associated with a database connection. Database adapters (or DAs for short) are available for a number of databases:

Oracle
: Oracle is a powerful and popular commercial relational database. This DA is written and commercially supported by Zope Corporation. Oracle can be purchased or evaluated from the Oracle Website.

Sybase
: Sybase is another popular commercial relational database. The Sybase DA is written and commercially supported by Zope Corporation. Sybase can be purchased or evaluated from the Sybase Website.

ODBC
: ODBC is a cross−platform, industry standard database protocol supported by many commercial and open source databases. The ODBC DA is written and commercially supported by Zope Corporation.

PostgreSQL
: PostgreSQL is a leading open source relational database. There are several database adapters for PostgreSQL including ZPoPy which is maintained by Zope community member Thierry Michel. You can find more information about PostgreSQL at the PostgreSQL web site.

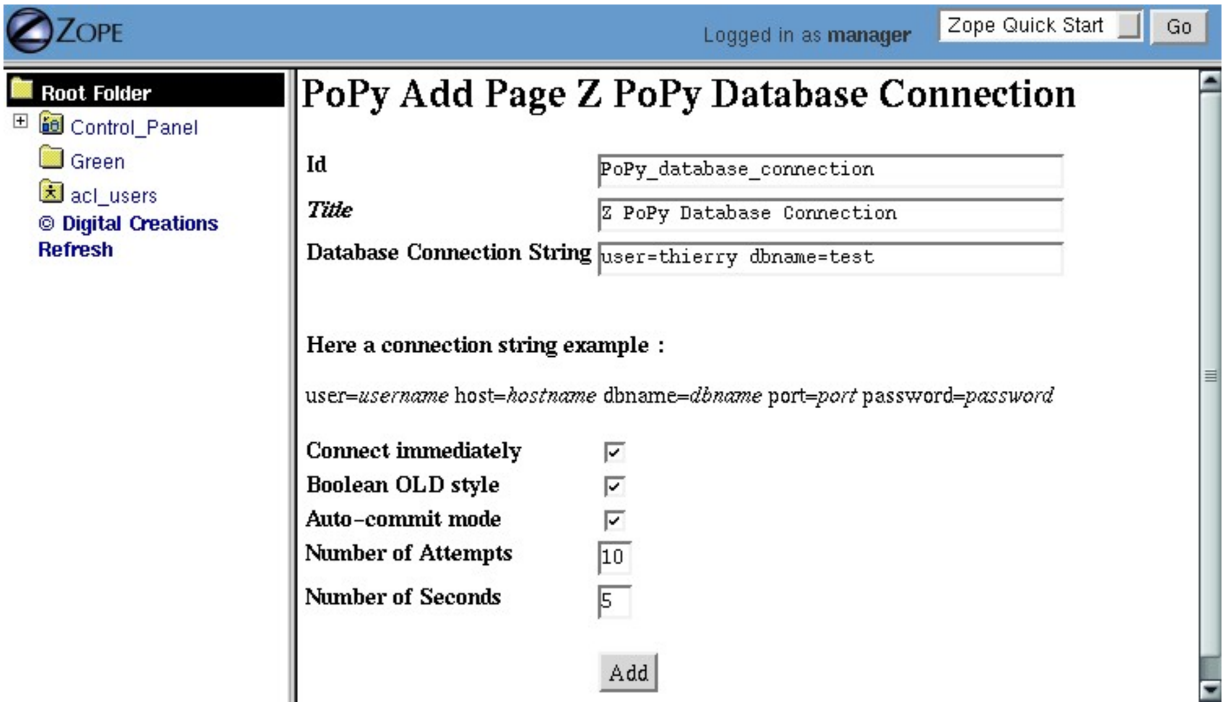
MySQL
: MySQL is a fast open source relational database. You can find more information about MySQL at the MySQL web site. The MySQL DA is maintained by Zope community member Monty Taylor.

Interbase
: Interbase is an open source relational database from Borland/Inprise. You can find more information about Interbase at the Borland web site. You may also be interested in FireBird which is a community maintained offshoot of Interbase. The Zope Interbase adapter is maintained by Zope community member Bob Tierney.

Gadfly
: Gadfly is a relational database written in Python by Aaron Waters. Gadfly is included with Zope for demonstration purposes and small data sets. Gadfly is fast, but is not intended for large amounts of information since it reads the entire database into memory. You can find out more about Gadfly at the Chordate website.

Other than Gadfly, all relational databases run as processes external to Zope. In fact, your relational database need not even run on the same machine as Zope, so long as Zope can connect to the machine that the database is running on. Installing and setting up relational databases is beyond the scope of this book. All of the relational databases mentioned have their own installation and configuration documentation that you should consult for specific details.

Because Gadfly runs inside Zope, you do not need to specify any connection information for Zope to find the database. Since all other kinds of databases run externally to Zope, they require you to specify how to connect to the database. This specification, called a *connection string*, is different for each kind of database. For example, Figure 10−2 shows the PostgreSQL database connection add form.



PoPy Add Page Z PoPy Database Connection

Id	PoPy_database_connection
Title	Z PoPy Database Connection
Database Connection String	user=thierry dbname=test

Here a connection string example :

user=*username* host=*hostname* dbname=*dbname* port=*port* password=*password*

Connect immediately	✓
Boolean OLD style	✓
Auto-commit mode	✓
Number of Attempts	10
Number of Seconds	5

Figure 10−2 PostgreSQL Database Connection

For PostgreSQL, the connection string format is shown above in Figure 10−2.

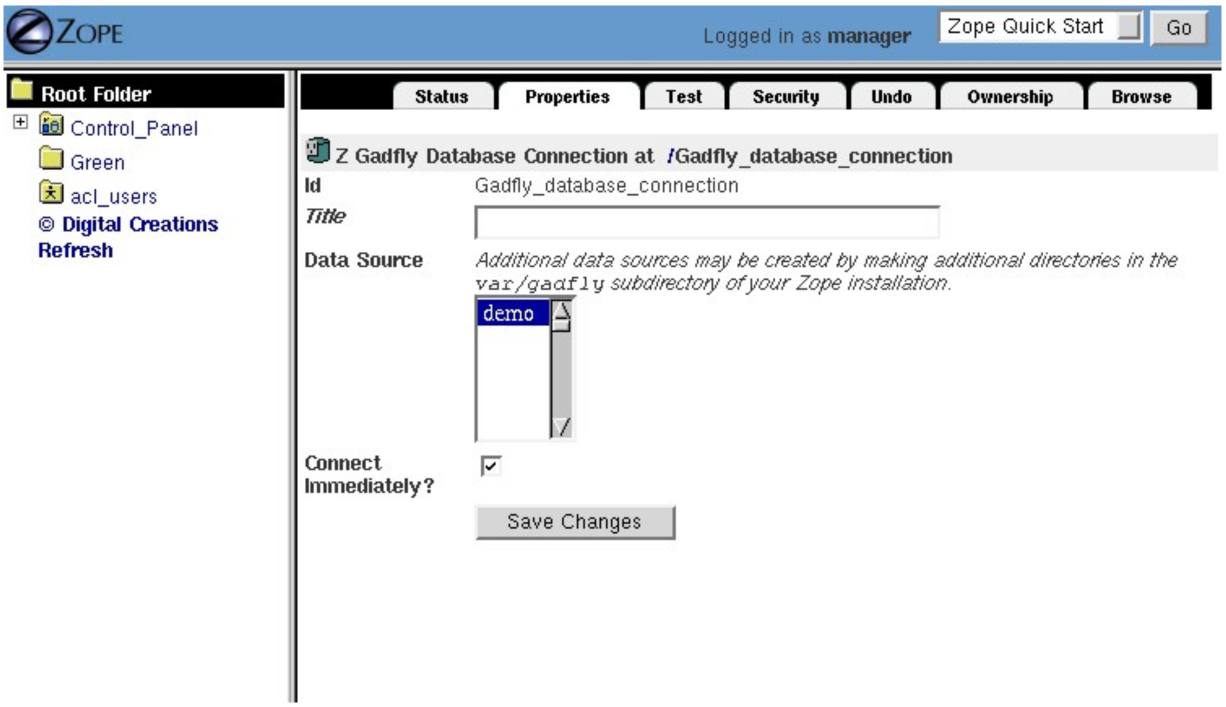
In order to use your relational database of choice from Zope, you must download and install the database adapter for your specific relational database. Database adapters can be downloaded from the Products section of Zope.org The exception to this is Gadfly, which is included with Zope. All the examples in this chapter use Gadfly, but the procedures described apply to all databases.

After installing the database adapter product for your database, you can create a new database connection by selecting it from the *Add List*. All database connections are fairly similar. Select the *Z Gadfly Database Connection* from the add list. This will take you to the add form for a Gadfly database connection.

Select the *Demo* data source, specify *Gadfly_database_connection* for the id, and click the *Add* button. This will create a new Gadfly Database Connection. Select the new connection by clicking on it.

You are looking at the *Status* view of the Gadfly Database Connection. This view tells you if you are connected to the database, and there is a button to connect or disconnect. In general Zope will manage the connection to your database for you so there is little reason to manually control the connection. For Gadfly connecting and disconnecting are meaningless, but for external databases you may wish to connect or disconnect manually to do database maintenance.

The next view is the *Properties* view. This view shows you the data source and other properties of the Database Connection. This is useful if you want to move your Database Connection from one data source to another. Figure 10−3 shows the *Properties* view.



Z Gadfly Database Connection at /Gadfly_database_connection

Id Gadfly_database_connection

Title

Data Source *Additional data sources may be created by making additional directories in the* `var/gadfly` *subdirectory of your Zope installation.*

demo

Connect Immediately? ✓

Figure 10−3 The Properties view

You can test your connection to a database by going to the *Test* view. This view lets you type SQL code directly and run it on your database. This view is just for testing your database and issuing one time SQL commands (like creating tables). This is *not* the place where you will enter most of your SQL code. SQL commands reside in Z SQL Methods which are discussed later in this chapter.

Let's create a table in your database to use in this chapter's examples. The *Test* view of the Database Connection allows you to send SQL statements directly to your database. You can create tables by typing SQL code directly into the *Test* view; there is no need to use a SQL Method to create tables. Create a table called *employees* with the following SQL code:

```
CREATE TABLE employees
(
emp_id integer,
first varchar,
last varchar,
salary float
)
```

Click the *Submit Query* button to run the SQL command. Zope should return a confirmation screen that tells you what SQL code was run and the results if any.

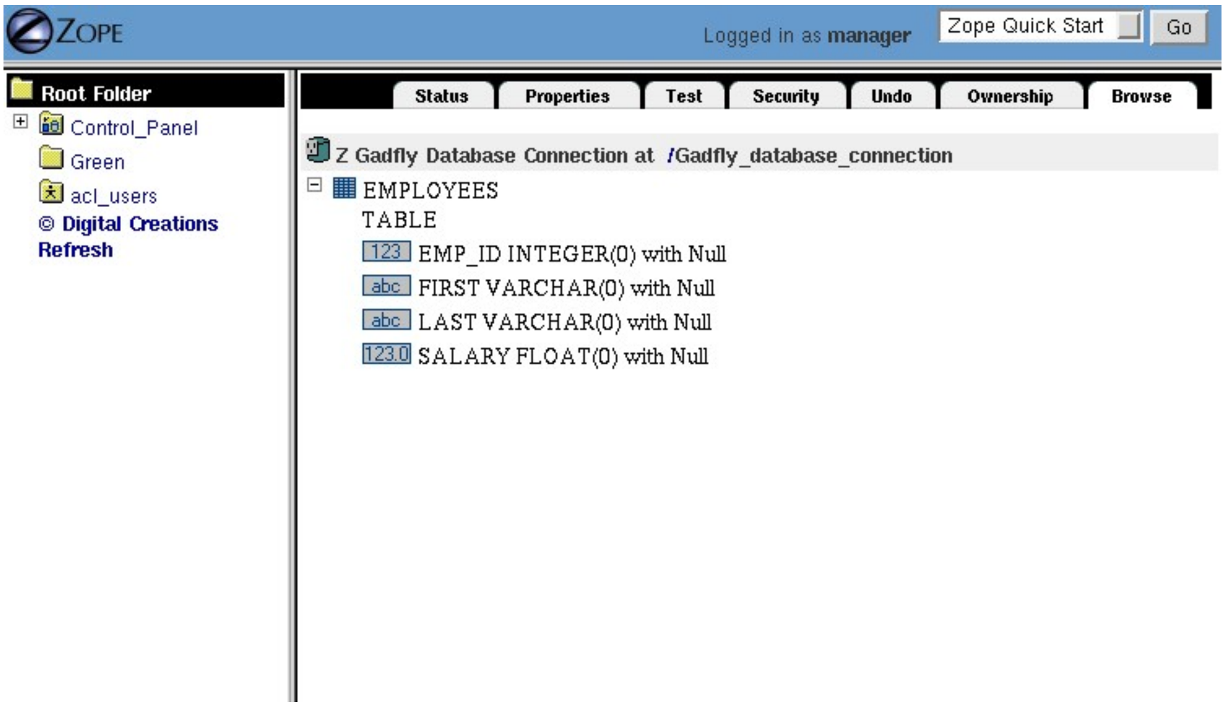
The SQL used here may differ depending on your database. For the exact details of creating tables with your database, check the user documentation from your specific database vendor.

This SQL will create a new table in your Gadfly database called *employees*. This table will have four columns, *emp_id*, *first*, *last* and *salary*. The first column is the employee id, which is a unique number that identifies the employee. The next two columns have the type *varchar* which is similar to a string. The *salary* column has the type *float* which holds a floating point number. Every database supports different kinds of types, so consult your documentation to find out what kind of types your database supports.

To ensure that the employee id is a unique number you can create an index on your table. Type the following SQL code in the *Test* view:

```
CREATE UNIQUE INDEX emp_id ON employees
(
emp_id
)
```

Now you have a table and an index. To examine your table, go to the *Browse* view. This view lets you view your database's tables and their schemas. Here, you can see that there is an *employees* table, and if you click on the *plus symbol*, the table expands to show four columns, *emp_id*, *first*, *last* and *salary* as shown in Figure 10−4.



Z Gadfly Database Connection at /Gadfly_database_connection
EMPLOYEES
TABLE
EMP_ID INTEGER(0) with Null
FIRST VARCHAR(0) with Null
LAST VARCHAR(0) with Null
SALARY FLOAT(0) with Null

Figure 10−4 Browsing the Database Connection

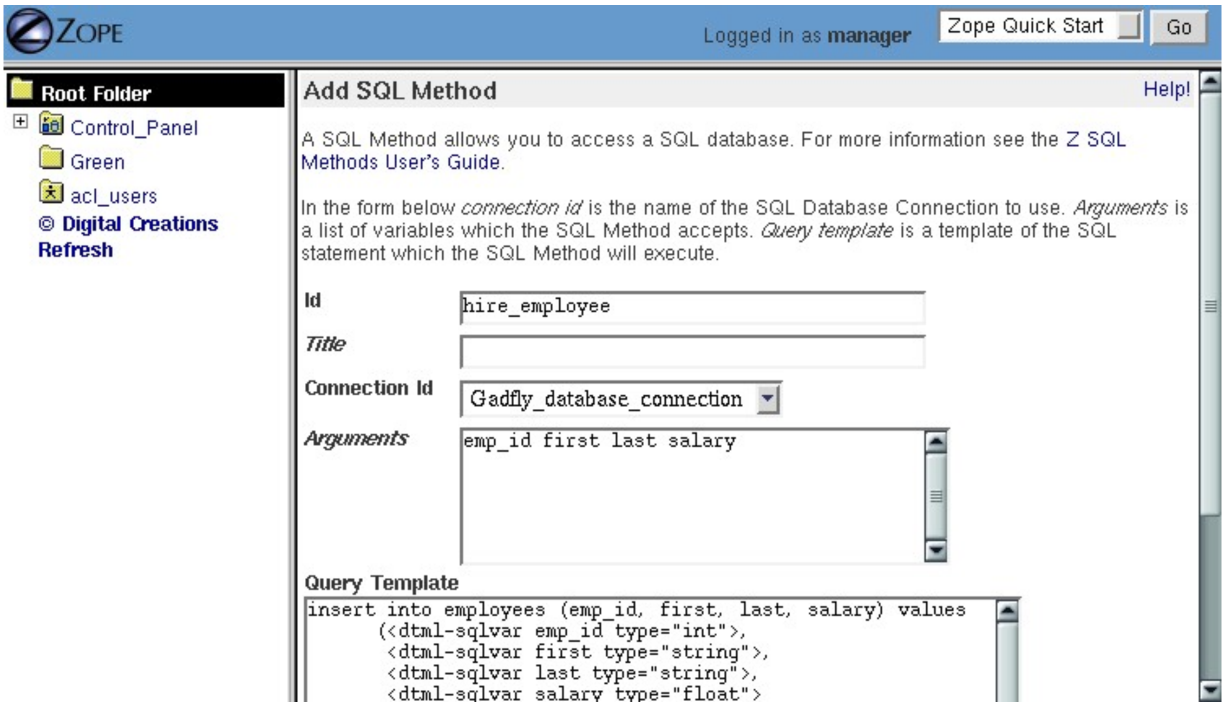
This information is very useful when creating complex SQL applications with lots of large tables as it lets you discover the schemas of your tables. Not all databases support browsing of tables.

Now that you've created a database connection and have defined a table, you can create Z SQL Methods to operate on your database.

Using Z SQL Methods

Z SQL Methods are Zope object that execute SQL code through a Database Connection. All Z SQL Methods must be associated with a Database Connection. Z SQL Methods can both query databases and change data. Z SQL Methods can also contain more than one SQL command.

Next, you need to create a new Z SQL Method called *hire_employee* that inserts a new employee in the *employees* table. When a new employee is hired this method is called and a new record is inserted in the *employees* table that contains the information about the new employee. Select *Z SQL Method* from the *Add List*. This will take you to the add form for Z SQL Methods, as shown in Figure 10−5.



Add SQL Method

A SQL Method allows you to access a SQL database. For more information see the Z SQL Methods User's Guide.

In the form below *connection id* is the name of the SQL Database Connection to use. *Arguments* is a list of variables which the SQL Method accepts. *Query template* is a template of the SQL statement which the SQL Method will execute.

Id	hire_employee
Title	
Connection Id	Gadfly_database_connection
Arguments	emp_id first last salary

Query Template

```
insert into employees (emp_id, first, last, salary) values
       (<dtml-sqlvar emp_id type="int">,
        <dtml-sqlvar first type="string">,
        <dtml-sqlvar last type="string">,
        <dtml-sqlvar salary type="float">
```

Figure 10−5 The Add form for Z SQL Methods

As usual, you must specify an *id* and *title* for the Z SQL Method. In addition you need to select a Database Connection to use with this Z SQL Methods. Give this new method the id *hire_employee* and select the *Gadfly_database_connection* that you created in the last section.

Next you can specify *arguments* to the Z SQL Method. Just like Scripts, Z SQL Methods can take arguments. Arguments are used to construct SQL statements. In this case your method needs four arguments, the employee id number, the first name, the last name and the employee's salary. Type "emp_id first last salary" into the *Arguments* field. You can put each argument on its own line, or you can put more than one argument on the same line separated by spaces. You can also provide default values for argument just like with Python Scripts. For example, `empid=100` gives the `empid` argument a default value of 100.

The last form field is the *Query template*. This field contains the SQL code that is executed when the Z SQL Method is called. In this field, enter the following code:

```
insert into employees (emp_id, first, last, salary) values
(<dtml-sqlvar emp_id type="int">,
 <dtml-sqlvar first type="string">,
 <dtml-sqlvar last type="string">,
 <dtml-sqlvar salary type="float">
)
```

Notice that this SQL code also contains DTML. The DTML code in this template is used to insert the values of the arguments into the SQL code that gets executed on your database. So, if the *emp_id* argument had the value *42*, the *first* argument had the value *Bob* your *last* argument had the value *Uncle* and the *salary* argument had the value *50000.00* then the query template would create the following SQL code:

```
insert into employees (emp_id, first, last, salary) values
(42,
 'Bob',
 'Uncle',
 50000.00
)
```

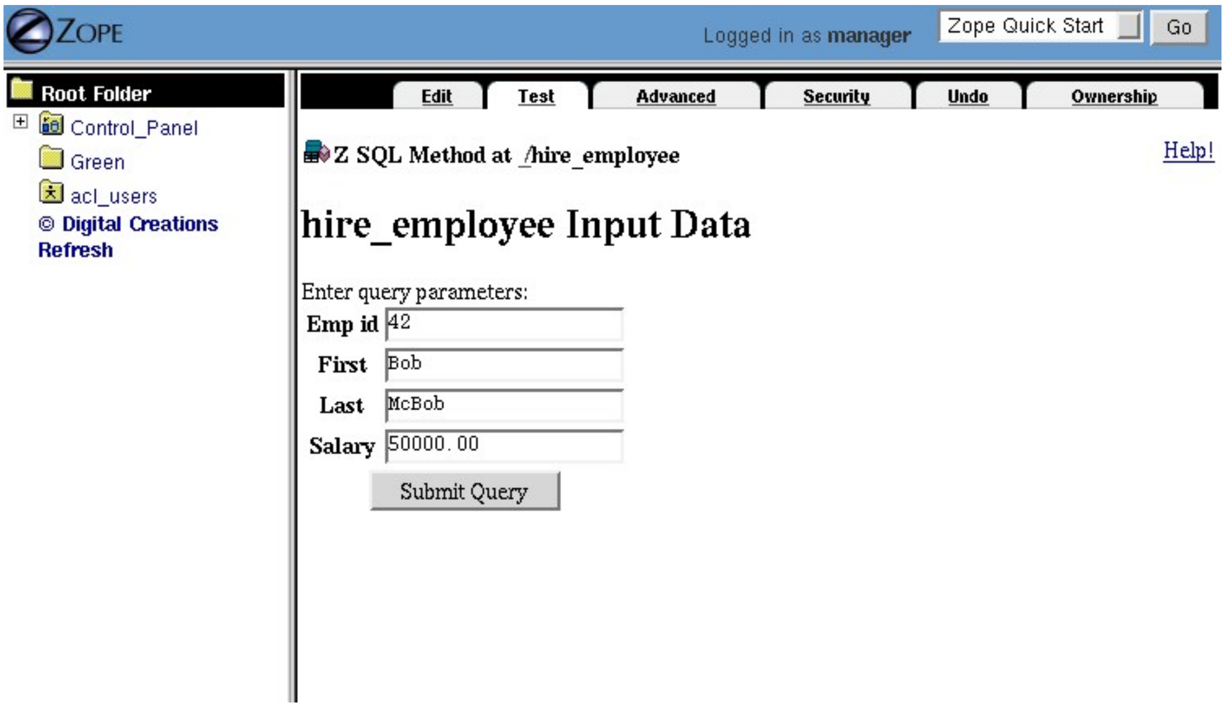
The query template and SQL−specific DTML tags are explained further in the next section.

You have your choice of three buttons to click to add your new Z SQL Method. The *Add* button will create the method and take you back to the folder containing the new method. The *Add and Edit* button will create the method and make it the currently selected object in the *Workspace*. The *Add and Test* button will create the method and take you to the method's *Test* view so you can test the new method. To add your new Z SQL Method, click the *Add* button.

Now you have a Z SQL Method that inserts new employees in the *employees* table. You'll need another Z SQL Method to query the table for employees. Create a new Z SQL Method with the id *list_all_employees*. It should have no arguments and contain the SQL code:

```
select * from employees
```

This simple SQL code selects all the rows from the *employees* table. Now you have two Z SQL Methods, one to insert new employees and one to view all of the employees in the database. Let's test your two new methods by inserting some new employees in the *employees* table and then listing them. To do this, click on the *hire_employee* Method and click the *Test* tab. This will take you to the *Test* view of the Method, as shown in Figure 10−6.



hire_employee Input Data

Enter query parameters:

Emp id	42
First	Bob
Last	McBob
Salary	50000.00

Submit Query

Figure 10−6 The hire_employee Test view

Here, you see a form with four input boxes, one for each argument to the *hire_employee* Z SQL Method. Zope automatically generates this form for you based on the arguments of your Z SQL Method. Because the *hire_employee* Method has four arguments, Zope creates this form with four input boxes. You can test the method by entering an employee number, a first name, a last name, and a salary for your new employee. Enter the employee id "42", "Bob" for the first name, "McBob" for the last name and a salary of "50000.00". Then click the *Test* button. You will then see the results of your test.

The screen says *This statement returned no results*. This is because the *hire_employee* method only inserts new information in the table, it does not select any information out of the table, so no records were returned. The screen also shows you how the query template get rendered into SQL. As expected, the *sqlvar* DTML tags rendered the four arguments into valid SQL code that your database executed. You can add as many employees as you'd like by repeatedly testing this method.

To verify that the information you added is being inserted into the table, select the *list_all_employees* Z SQL Method and click on its *Test* tab.

This view says *This query requires no input*, indicating the *list_all_employees* does not have any argument and thus, requires no input to execute. Click on the *Submit Query* button to test the method.

The *list_all_employees* method returns the contents of your *employees* table. You can see all the new employees that you added. Zope automatically generates this tabular report screen for you. Next we'll show how you can create your own user interface to your Z SQL Methods to integrate them into your web site.

Calling Z SQL Methods

Querying a relational database returns a sequence of results. The items in the sequence are called *result rows*. SQL query results are always a sequence. Even if the SQL query returns only one row, that row is the only item contained in a list of results. Hence, Z SQL Methods *always* return a sequence of results which contains zero or more results records.

The items in the sequence of results returned by a Z SQL Method are called *Result objects*. Result objects can be thought of as rows from the database table turned into Zope objects. These objects have attributes that match the schema of the database results.

An important difference between result objects and other Zope objects is that result objects do not get created and permanently added to Zope. Result objects are not persistent. They exist for only a short period of time; just long enough for you to use them in a result page or to use their data for some other purpose. As soon as you are done with a request that uses result objects they go away, and the next time you call a Z SQL Method you get a new set of fresh result objects.

Result objects can be used from DTML to display the results of calling a Z SQL Method. For example, add a new DTML Method to your site called *listEmployees* with the following DTML content:

```
<dtml-var standard_html_header>

  <ul>
  <dtml-in list_all_employees>
    <li><dtml-var emp_id>: <dtml-var last>, <dtml-var first>
      makes <dtml-var salary fmt=dollars-and-cents> a year.
    </li>
  </dtml-in>
  </ul>

<dtml-var standard_html_footer>
```

This method calls the *list_all_employees* Z SQL Method from DTML. The *in* tag is used to iterate over each Result object returned by the *list_all_employees* Z SQL Method. Z SQL Methods always return a list of objects, so you will almost certainly use them from the DTML *in* tag unless you are not interested in the results or if the SQL code will never return any results, like *hire_employee*.

The body of the *in* tag is a template that defines what gets rendered for each Result object in the sequence returned by *list_all_employees*. In the case of a table with three employees in it, *listEmployees* might return HTML that looks like this:

```
<html>
  <body>

  <ul>
    <li>42: Roberts, Bob
      makes $50,000 a year.
    </li>
    <li>101: leCat, Cheeta
      makes $100,000 a year.
    </li>
    <li>99: Junglewoman, Jane
      makes $100,001 a year.
    </li>
  </ul>

  </body>
</html>
```

The *in* tag rendered an HTML list item for each Result object returned by *list_all_employees*.

Next we'll look at how to create user interfaces in order to collect data and pass it to Z SQL Methods.

Providing Arguments to Z SQL Methods

So far, you have the ability to display employees with the the *listEmployees* DTML Method which calls the *list_all_employees* Z SQL Method. Now let's look at how to build a user interface for the *hire_employee* Z SQL Method. Recall that the *hire_employee* accepts four arguments, *emp_id*, *first*, *last*, and *salary*. The *Test* tab on the *hire_employee* method lets you call this method, but this is not very useful for integrating into a web application. You need to create your own input form for your Z SQL Method or call it manually from your application.

The Z Search Interface can create an input form for you automatically. In Chapter 11, "Searching and Categorizing Content", you used the Z Search Interface to build a form/action pair of methods that automatically generated an HTML search form and report screen that queried the Catalog and returned results. The Z Search Interface also works with Z SQL Methods to build a similar set of search/result screens.

Select *Z Search Interface* from the add list and specify *hire_employee* as the *Searchable object*. Enter the value "hireEmployee" for the *Report Id* and "hireEmployeeForm" for the *Search Id* and click *Add*.

Click on the newly created *hireEmployeeForm* and click the *View* tab. Enter an employee_id, a first name, a last name, and salary for a new employee and click *Submit*. Zope returns a screen that says "There was no data matching this query". Because the report form generated by the Z Search Interface is meant to display the result of a Z SQL Method, and the *hire_employee* Z SQL Method does not return any results; it just inserts a new row in the table. Edit the *hireEmployee* DTML Method a little to make it more informative. Select the *hireEmployee* Method. It should contain the following long stretch of DTML:

```
<dtml-var standard_html_header>

<dtml-in hire_employee size=50 start=query_start>

   <dtml-if sequence-start>

      <dtml-if previous-sequence>

        <a href="<dtml-var URL><dtml-var sequence-query
                 >query_start=<dtml-var
                 previous-sequence-start-number>">
        (Previous <dtml-var previous-sequence-size> results)
```

```
              </a>

            </dtml-if previous-sequence>

            <table border>
              <tr>
              </tr>

         </dtml-if sequence-start>

              <tr>
              </tr>

         <dtml-if sequence-end>

            </table>
            <dtml-if next-sequence>

               <a href="<dtml-var URL><dtml-var sequence-query
                  >query_start=<dtml-var
                  next-sequence-start-number>">
               (Next <dtml-var next-sequence-size> results)
               </a>

            </dtml-if next-sequence>

         </dtml-if sequence-end>

      <dtml-else>

        There was no data matching this <dtml-var title_or_id> query.

      </dtml-in>

      <dtml-var standard_html_footer>
```

This is a pretty big piece of DTML! All of this DTML is meant to dynamically build a batch−oriented tabular result form. Since we don't need this, let's change the *hireEmployee* method to be much simpler:

```
<dtml-var standard_html_header>

<dtml-call hire_employee>

<h1>Employee <dtml-var first> <dtml-var last> was Hired!</h1>

<p><a href="listEmployees">List Employees</a></p>

<p><a href="hireEmployeeForm">Back to hiring</a></p>

<dtml-var standard_html_footer>
```

Now view *hireEmployeeForm* and hire another new employee. Notice how the *hire_employee* method is called from the DTML *call* tag. This is because we know there is no output from the *hire_employee* method. Since there are no results to iterate over, the method does not need to be called with the *in* tag. It can be called simply with the *call* tag.

Now you have a complete user interface for hiring new employees. Using Zope's security system, you can now restrict access to this method to only a certain group of users whom you want to have permission to hire new employees. Keep in mind, the search and report screens generated by the Z Search Interface are just guidelines that you can easily customize to suite your needs.

Next we'll take a closer look at precisely controlling SQL queries. You've already seen how Z SQL Methods allow you to create basic SQL query templates. In the next section you'll learn how to make the most of your query templates.

Dynamic SQL Queries

A Z SQL Method query template can contain DTML that is evaluated when the method is called. This DTML can be used to modify the SQL code that is executed by the relational database. Several SQL specific DTML tags exist to assist you in the construction of complex SQL queries. In the next sections you'll learn about the *sqlvar*, *sqltest*, and *sqlgroup* tags.

Inserting Arguments with the *Sqlvar* Tag

It's pretty important to make sure you insert the right kind of data into a column in a database. You database will complain if you try to use the string "12" where the integer 12 is expected. SQL requires that different types be quoted differently. To make matters worse, different databases have different quoting rules.

In addition to avoiding errors, SQL quoting is important for security. Suppose you had a query that makes a select:

```
select * from employees
  where emp_id=<dtml-var emp_id>
```

This query is unsafe since someone could slip SQL code into your query by entering something like *12; drop table employees* as an *emp_id*. To avoid this problem you need to make sure that your variables are properly quoted. The *sqlvar* tag does this for you. Here is a safe version of the above query that uses *sqlvar*:

```
select * from employees
  where emp_id=<dtml-sqlvar emp_id type=int>
```

The *sqlvar* tag operates similarly to the regular DTML *var* tag in that it inserts values. However it has some tag attributes targeted at SQL type quoting, and dealing with null values. The *sqlvar* tag accepts a number of arguments:

name
: The *name* argument is identical to the name argument for the *var* tag. This is the name of a Zope variable or Z SQL Method argument. The value of the variable or argument is inserted into the SQL Query Template. A *name* argument is required, but the "name=" prefix may be omitted.

type
: The *type* argument determines the way the *sqlvar* tag should format the value of the variable or argument being inserted in the query template. Valid values for type are *string*, *int*, *float*, or *nb*. *nb* stands for non−blank and means a string with at least one character in it. The *sqlvar* tag *type* argument is required.

optional
: The *optional* argument tells the *sqlvar* tag that the variable or argument can be absent or be a null value. If the variable or argument does not exist or is a null value, the *sqlvar* tag does not try to render it. The *sqlvar* tag *optional* argument is optional.

The *type* argument is the key feature of the *sqlvar* tag. It is responsible for correctly quoting the inserted variable. See Appendix A for complete coverage of the *sqlvar* tag.

You should always use the *sqlvar* tag instead of the *var* tag when inserting variables into a SQL code since it correctly quotes variables and keeps your SQL safe.

Equality Comparisons with the *Sqltest* Tag

Many SQL queries involve equality comparison operations. These are queries that ask for all values from the table that are in some kind of equality relationship with the input. For example, you may wish to query the *employees* table for all employees with a salary *greater than* a certain value.

To see how this is done, create a new Z SQL Method named *employees_paid_more_than*. Give it one argument, *salary*, and the following SQL template:

```
select * from employees
  where <dtml-sqltest salary op=gt type=float>
```

Now click *Add and Test*. The *op* tag attribute is set to *gt*, which stands for *greater than*. This Z SQL Method will only return records of employees that have a higher salary than what you enter in this input form. The *sqltest* builds the SQL syntax necessary to safely compare the input to the table column. Type "10000" into the *salary* input and click the *Test* button. As you can see the *sqltest* tag renders this SQL code:

```
select * from employees
  where salary > 10000
```

The *sqltest* tag renders these comparisons to SQL taking into account the type of the variable and the particularities of the database. The *sqltest* tag accepts the following tag parameters:

name
: The name of the variable to insert.

type
: The data type of the value to be inserted. This attribute is required and may be one of *string*, *int*, *float*, or *nb*. The nb data type stands for "not blank" and indicates a string that must have a length that is greater than 0. When using the nb type, the *sqltest* tag will not render if the variable is an empty string.

column
: The name of the SQL column, if different than the *name* attribute.

multiple
: A flag indicating whether multiple values may be provided. This lets you test if a column is in a set of variables. For example when *name* is a list of strings "Bob" , "Billy" , `<dtml-sqltest name type="string" multiple>` renders to this SQL: `name in ("Bob", "Billy")`.

optional
: A flag indicating if the test is optional. If the test is optional and no value is provided for a variable then no text is inserted. If the value is an empty string, then no text will be inserted only if the type is *nb*.

op
: A parameter used to choose the comparison operator that is rendered. The comparisons are: *eq* (equal to), *gt* (greater than), *lt* (less than), *ge* (greater than or equal to), *le* (less than or equal to), and *ne* (not equal to).

See Appendix A for more information on the *sqltest* tag. If your database supports additional comparison operators such as *like* you can use them with *sqlvar*. For example if *name* is the string "Mc%", the SQL code:

```
<dtml-sqltest name type="string" op="like">
```

would render to:

```
name like 'Mc%'
```

The *sqltest* tag helps you build correct SQL queries. In general your queries will be more flexible and work better with different types of input and different database if you use *sqltest* rather than hand coding comparisons.

Creating Complex Queries with the *Sqlgroup* Tag

The *sqlgroup* tag lets you create SQL queries that support a variable number of arguments. Based on the arguments specified, SQL queries can be made more specific by providing more arguments, or less specific by providing less or no arguments.

Here is an example of an unqualified SQL query:

```
select * from employees
```

Here is an example of a SQL query qualified by salary:

```
select * from employees
where(
  salary > 100000.00
)
```

Here is an example of a SQL query qualified by salary and first name:

```
select * from employees
where(
  salary > 100000.00
  and
  first in ('Jane', 'Cheetah', 'Guido')
)
```

Here is an example of a SQL query qualified by a first and a last name:

```
select * from employees
where(
  first = 'Old'
  and
  last = 'McDonald'
)
```

All three of these queries can be accomplished with one Z SQL Method that creates more specific SQL queries as more arguments are specified. The following SQL template can build all three of the above queries:

```
select * from employees
<dtml-sqlgroup where>
  <dtml-sqltest salary op=gt type=float optional>
<dtml-and>
  <dtml-sqltest first op=eq type=nb multiple optional>
<dtml-and>
  <dtml-sqltest last  op=eq type=nb multiple optional>
</dtml-sqlgroup>
```

The *sqlgroup* tag renders the string *where* if the contents of the tag body contain any text and builds the qualifying statements into the query. This *sqlgroup* tag will not render the *where* clause if no arguments are present.

The *sqlgroup* tag consists of three blocks separated by *and* tags. These tags insert the string *and* if the enclosing blocks render a value. This way the correct number of *ands* are included in the query. As more arguments are specified, more qualifying statements are added to the query. In this example, qualifying statements restricted the search with *and* tags, but *or* tags can also be used to expand the search.

This example also illustrates *multiple* attribute on *sqltest* tags. If the value for *first* or *last* is a list, then the right SQL is rendered to specify a group of values instead of a single value.

You can also nest *sqlgroup* tags. For example:

```
select * from employees
<dtml-sqlgroup where>
  <dtml-sqlgroup>
     <dtml-sqltest first op=like type=nb>
  <dtml-and>
     <dtml-sqltest last op=like type=nb>
  </dtml-sqlgroup>
<dtml-or>
  <dtml-sqltest salary op=gt type=float>
</dtml-sqlgroup>
```

Given sample arguments, this template renders to SQL like so:

```
select * from employees
where
( (first like 'A%'
   and
   last like 'Smith'
  )
  or
  salary > 20000.0
)
```

You can construct very complex SQL statements with the *sqlgroup* tag. For simple SQL code you won't need to use the *sqlgroup* tag. However, if you find yourself creating a number of different but related Z SQL Methods you should see if you can't accomplish the same thing with one method that uses the *sqlgroup* tag.

Advanced Techniques

So far you've seen how to connect to a relational database, send it queries and commands, and create a user interface. These are the basics of relational database conductivity in Zope.

In the following sections you'll see how to integrate your relational queries more closely with Zope and enhance performance. We'll start by looking at how to pass arguments to Z SQL Methods both explicitly and by acquisition. Then you'll find out how you can call Z SQL Methods directly from URLs using traversal to result objects. Next you'll find out how to make results objects more powerful by binding them to classes. Finally we'll look at caching to improve performance and how Zope handles database transactions.

Calling Z SQL Methods with Explicit Arguments

If you call a Z SQL Method without argument from DTML, the arguments are automatically collected from the environment. This is the technique that we have used so far in this chapter. It works well when you want to query a database from a search form, but sometimes you want to manually or programmatically query a database. Z SQL Methods can be called with explicit arguments from DTML or Python. For example, to query the *employee_by_id* Z SQL Method manually, the following DTML can be used:

```
<dtml-var standard_html_header>
```

```
  <dtml-in expr="employee_by_id(emp_id=42)">
    <h1><dtml-var last>, <dtml-var first></h1>

    <p><dtml-var first>'s employee id is <dtml-var emp_id>.  <dtml-var
    first> makes <dtml-var salary fmt=dollars-and-cents> per year.</p>
  </dtml-in>

<dtml-var standard_html_footer>
```

Remember, the *employee_by_id* method returns only one record, so the body of the *in* tag in this method will execute only once. In the example you calling the Z SQL Method like any other method and passing it a keyword argument for *emp_id*. The same can be done easily from Python:

```
## Script (Python) "join_name"
##parameters=id
##
for result in context.employee_by_id(emp_id=id):
    return result.last + ', ' + result.first
```

This script accepts an *id* argument and passes it to *employee_by_id* as the *emp_id* argument. It then iterates over the single result and joins the last name and the first name with a comma.

You can provide more control over your relational data by calling Z SQL Methods with explicit arguments. It's also worth noting that from DTML and Python Z SQL Methods can be called with explicit arguments just like you call other Zope methods.

Acquiring Arguments from other Objects

Z SQL can acquire information from other objects and be used to modify the SQL query. Consider Figure 10−7, which shows a collection of Folders in a organization's web site.

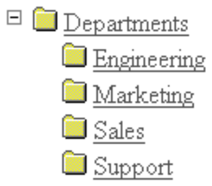


Figure 10−7 Folder structure of an organizational web site

Suppose each department folder has a *department_id* string property that identifies the accounting ledger id for that department. This property could be used by a shared Z SQL Method to query information for just that department. To illustrate, create various nested folders with different *department_id* string properties and then create a Z SQL Method with the id *requisition_something* in the root folder that takes three arguments, *description*, *quantity*, and *unit_cost*. and the following query template:

```
INSERT INTO requisitions
  (
    department_id, description, quantity, unit_cost
  )
VALUES
  (
    <dtml-sqlvar department_id type=string>,
    <dtml-sqlvar description type=string>,
    <dtml-sqlvar quantity type=int>,
    <dtml-sqlvar unit_cost type=float>
  )
```

Now, create a Z Search Interface with a *Search Id* of "requisitionSomethingForm" and the *Report id* of "requisitionSomething". Select the *requisition_something* Z SQL Method as the *Searchable Object* and click *Add*.

Edit the *requisitionSomethingForm* and remove the first input box for the *department_id* field. We don't want the value of *department_id* to come from the form, we want it to come from a property that is acquired.

Now, you should be able to go to a URL like:

```
http://example.org/Departments/Support/requisitionSomethingForm
```

and requisition some punching bags for the Support department. Alternatively, you could go to:

```
http://example.org/Departments/Sales/requisitionSomethingForm
```

And requisition some tacky rubber key−chains with your logo on them for the Sales department. Using Zope's security system as described in Chapter 7, "Users and Security", you can now restrict access to these forms so personnel from departments can requisition items just for their department and not any other.

The interesting thing about this example is that *department_id* was not one of the arguments provided to the query. Instead of getting the value of this variable from an argument, it *acquires* the value from the folder where the Z SQL Method is accessed. In the case of the above URLs, the *requisition_something* Z SQL Method acquires the value from the *Sales* and *Support* folders. This allows you to tailor SQL queries for different purposes. All the departments can share a query but it is customized for each department.

By using acquisition and explicit argument passing you can tailor your SQL queries to your web application.

Traversing to Result Objects

So far you've provided arguments to Z SQL Methods from web forms, explicit argument, and acquisition. You can also provide arguments to Z SQL Methods by calling them from the web with special URLs. This is called *traversing* to results objects. Using this technique you can walk directly up to result objects using URLs.

In order to traverse to result objects with URLs, you must be able to ensure that the SQL Method will return only one result object given one argument. For example, create a new Z SQL Method named *employee_by_id* that accepts one argument, *emp_id*, and has the following SQL Template:

```
select * from employees where
  <dtml−sqltest emp_id op=eq type=int>
```

This method selects one employee out of the *employees* table based on their employee id. Since each employee has a unique id, only one record will be returned. Relational databases can provide these kinds of uniqueness guarantees.

Zope provides a special URL syntax to access ZSQL Methods that always return a single result. The URL consists of the URL of the ZSQL Method followed by the argument name followed by the argument value. For example, *http://localhost:8080/employee_by_id/emp_id/42*. Note, this URL will return a single result object where as if you queried the ZSQL Method from DTML and passed it a single argument it would return a list of results that happend to only have one item in it.

Unfortunately the result object you get with this URL is not very interesting to look at. It has no way to display itself in HTML. You still need to display the result object. To do this, you can call a DTML Method on the result object. This can be done using the normal URL acquisition rules described in Chapter 10, "Advanced Zope Scripting". For example, consider the following URL:

```
http://localhost:8080/employee_by_id/emp_id/42/viewEmployee
```

Here we see the *employee_by_id* Z SQL Method being passed the *emp_id* argument by URL. The *viewEmployee* method is then called on the result object. Let's create a *viewEmployee* DTML Method and try it out. Create a new DTML Method named *viewEmployee* and give it the following content:

```
<dtml-var standard_html_header>

  <h1><dtml-var last>, <dtml-var first></h1>

  <p><dtml-var first>'s employee id is <dtml-var emp_id>.  <dtml-var
  first> makes <dtml-var salary fmt=dollars-and-cents> per year.</p>

<dtml-var standard_html_footer>
```

Now when you go to the URL *http://localhost:8080/employee_by_id/emp_id/42/viewEmployee* the *viewEmployee* DTML Method is bound the result object that is returned by *employee_by_id*. The *viewEmployee* method can be used as a generic template used by many different Z SQL Methods that all return employee records.

Since the *employee_by_id* method only accepts one argument, it isn't even necessary to specify *emp_id* in the URL to qualify the numeric argument. If your Z SQL Method has one argument, then you can configure the Z SQL Method to accept only one extra path element argument instead of a pair of arguments. This example can be simplified even more by selecting the *employee_by_id* Z SQL Method and clicking on the *Advanced* tab. Here, you can see a check box called *Allow "Simple" Direct Traversal*. Check this box and click *Change*. Now, you can browse employee records with simpler URLs like *http://localhost:8080/employee_by_id/42/viewEmployee*. Notice how no *emp_id* qualifier is declared in the URL.

Traversal gives you an easy way to provide arguments and bind methods to Z SQL Methods and their results. Next we'll show you how to bind whole classes to result objects to make them even more powerful.

Binding Classes to Result Objects

A result object has an attribute for each column in results row. However, result objects do not have any methods, just attributes.

There are two ways to bind a method to a Result object. As you saw in the previous section, you can bind DTML and other methods to Z SQL Method Result objects using traversal to the results object coupled with the normal URL based acquisition bind mechanism described in Chapter 10, "Advanced Zope Scripting". You can also bind methods to Result objects by defining a Python class that gets *mixed in* with the normal, simple Result object class. These classes are defined in the same location as External Methods in the filesystem, in Zope's *Extensions* directory. Python classes are collections of methods and attributes. By associating a class with a Result object, you can make the Result object have a rich API and user interface.

Classes used to bind methods and other class attributes to Result classes are called *Pluggable Brains*, or just *Brains*. Consider the example Python class:

```
class Employee:

  def fullName(self):
    """ The full name in the form 'John Doe' """
```

```
        return self.first + ' ' + self.last
```

When result objects with this Brains class are created as the result of a Z SQL Method query, the Results objects will have *Employee* as a base class. This means that the record objects will have all the methods defined in the *Employee* class, giving them behavior, as well as data.

To use this class, create the above class in the *Employee.py* file in the *Extensions* directory. Go the *Advanced* tab of the *employee_by_id* Z SQL Method and enter *Employee* in the *Class Name* field, and *Employee* in the *Class File* field and click *Save Changes*. Now you can edit the *employeeView* DTML Method to contain:

```
<dtml-var standard_html_header>

  <h1><dtml-var fullName></h1>

  <p><dtml-var first>'s employee id is <dtml-var emp_id>.  <dtml-var
  first> makes <dtml-var salary fmt=dollars-and-cents> per year.</p>

<dtml-var standard_html_footer>
```

Now when you go to the URL *http://localhost:8080/employee_by_id/42/viewEmployee* the *fullName* method is called by the *viewEmployee* DTML Method. The *fullName* method is defined in the *Employee* class of the *Employee* module and is bound to the result object returned by *employee_by_id*

Brains provide a very powerful facility which allows you to treat your relational data in a more object−centric way. For example, not only can you access the *fullName* method using direct traversal, but you can use it anywhere you handle result objects. For example:

```
<dtml-in employee_by_id>
  <dtml-var fullName>
</dtml-in>
```

For all practical purposes your Z SQL Method returns a sequence of smart objects, not just data.

This example only scratches the surface of what can be done with Brains classes. Python programming is beyond the scope of this book so we will only go a little farther here. However, you could create brains classes that accessed network resources, called other Z SQL Methods, performed all kinds of business logic.

Here's a more powerful example of brains. Suppose that you have an *managers* table to go with the *employees* table that you've used so far. Suppose also that you have a *manager_by_id* Z SQL Method that returns a manager id manager given an *emp_id* argument:

```
select manager_id from managers where
  <dtml-sqltest emp_id type=int op=eq>
```

You could use this Z SQL Method in your brains class like so:

```
class Employee:

    def manager(self):
        """
        Returns this employee's manager or None if the
        employee does not have a manager.
        """
        # Calls the manager_by_id Z SQL Method.
        records=self.manager_by_id(emp_id=self.emp_id)
        if records:
            manager_id=records[0].manager_id
            # Return an employee object by calling the
```

```
                    # employee_by_id Z SQL Method with the manager's emp_id
                    return self.employee_by_id(emp_id=manager_id)[0]
```

This `Employee` class shows how methods can use other Zope objects to weave together relational data to make it seem like a collection of objects. The `manager` method calls two Z SQL Methods, one to figure out the emp_id of the employee's manager, and another to return a new Result object representing the manager. You can now treat employee objects as though they have simple references to their manager objects. For example you could add something like this to the *viewEmployee* DTML Method:

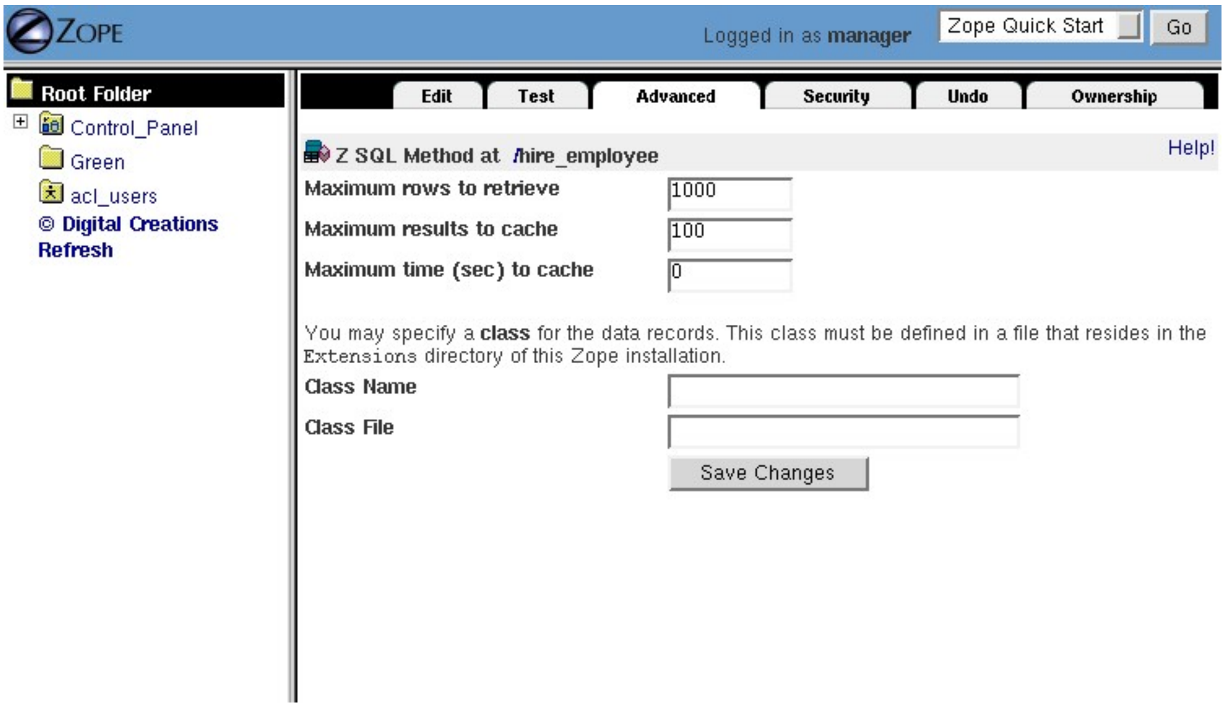
```
<dtml-if manager>
  <dtml-with manager>
    <p> My manager is <dtml-var first> <dtml-var last>.</p>
  </dtml-with>
</dtml-if>
```

As you can see brains can be both complex and powerful. When designing relational database applications you should try to keep things simple and add complexity slowly. It's important to make sure that your brains classes don't add lots of unneeded overhead.

Caching Results

You can increase the performance of your SQL queries with caching. Caching stores Z SQL Method results so that if you call the same method with the same arguments frequently, you won't have to connect to the database every time. Depending on your application, caching can dramatically improve performance.

To control caching, go to the *Advanced* tab of a SQL Method. You have three different cache controls as shown in Figure 10−8.



Z SQL Method at /hire_employee

Maximum rows to retrieve: 1000
Maximum results to cache: 100
Maximum time (sec) to cache: 0

You may specify a **class** for the data records. This class must be defined in a file that resides in the `Extensions` directory of this Zope installation.

Class Name:
Class File:

Figure 10−8 Caching controls for Z SQL Methods

The *Maximum number of rows received* field controls how much data to cache for each query. The *Maximum number of results to cache* field controls how many queries to cache. The *Maximum time (in seconds) to cache results* controls how long cached queries are saved for. In general, the larger you set these values the greater your performance increase, but the more memory Zope will consume. As with any performance tuning, you should experiment to find the optimum settings for your application.

In general you will want to set the maximum results to cache to just high enough and the maximum time to cache to be just long enough for your application. For site with few hits you should cache results for longer, and for sites with lots of hits you should cache results for a shorter period of time. For machines with lots of memory you should increase the number of cached results. To disable caching set the cache time to zero seconds. For most queries, the default value of 1000 for the maximum number of rows retrieved will be adequate. For extremely large queries you may have to increase this number in order to retrieve all your results.

Transactions

A transaction is a group of operations that can be undone all at once. As you saw in Chapter 1, "Introducing Zope", all changes done to Zope are done within transactions. Transactions ensure data integrity. When using a system that is not transactional and one of your web actions changes ten objects, and then fails to change the eleventh, then your data is now inconsistent. Transactions allow you to revert all the changes you made during a request if an error occurs.

Imagine the case where you have a web page that bills a customer for goods received. This page first deducts the goods from the inventory, and then deducts the amount from the customers account. If the second operations fails for some reason you want to make sure the change to the inventory doesn't take effect.

Most commercial and open source relational databases support transactions. If your relational database supports transactions, Zope will make sure that they are tied to Zope transactions. This ensures data integrity across both Zope and your relational database. If either Zope or the relational database aborts the transaction, the entire transaction is aborted.

Summary

Zope allows you to build web applications with relational databases. Unlike many web application servers, Zope has its own object database and does not require the use of relational databases to store information.

Zope lets you use relational data just like you use other Zope objects. You can connect your relational data to business logic with scripts and brains, you can query your relational data with Z SQL Methods and presentation tools like DTML, and your can even use advanced Zope features like URL traversal, acquisition, undo and security while working with relational data.

Chapter 13: Scalability and ZEO

When a web site gets more requests than it can handle it can become slow and unresponsive. In the worst case too many requests to a web site can cause the server to completely overload, stop handling requests and possibly even crash. This can be a problem for *any* kind of server application, not just Zope. The obvious solution to this problem is to use more than one computer, so in case one computer fails, another computer can continue to serve up your web site.

Using multiple computers has obvious benefits, but it also has some drawbacks. For example, if you had five computers running Zope then you must ensure that all five Zope installations have the same information on them. This is not a very hard task if you're the only user and you have only a few static objects, but for large organizations with thousands of rapidly changing objects, keeping five separate Zope installations synchronized manually would be a nightmare. To solve this problem, Zope Corporation created Zope Enterprise Objects, or ZEO. This chapter gives you a brief overview on installing ZEO, but there are many other options we don't cover. For more in−depth information, see the documentation that comes with the ZEO package, and also take a look at the ZEO discussion area.

What is ZEO?

ZEO is a system that allows you to run your site on more than one computer. This is often called *clustering* and *load balancing*. By running Zope on multiple computers, you can spread the requests evenly around and add more computers as the number of requests grows. Further, if one computer fails or crashes, other computers can still service requests while you fix the broken one.

ZEO runs Zope on multiple computers and takes care of making sure all the Zope installations share the exact same database at all times. ZEO uses a client/server architecture. The Zope installations on multiple computers are the *ZEO Clients*. All of the clients connect to one, central *ZEO Storage Server*, as shown in Figure 11−1.

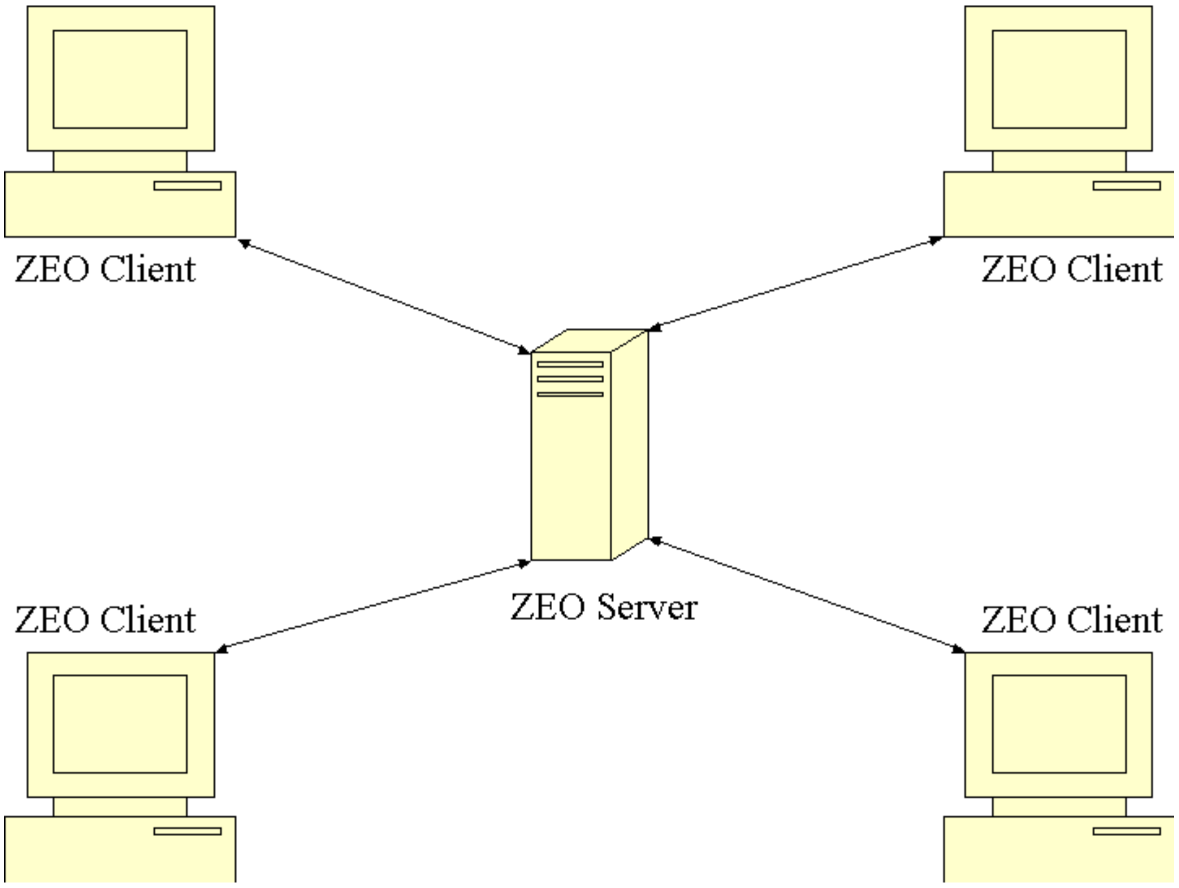


Figure 11−1 Simple ZEO illustration

The terminology can be a bit confusing, because normally you think of Zope as a server, not a client. When using ZEO, your Zope processes act as both servers (for web requests) and clients (for data from the ZEO server).

ZEO clients and servers communicate using standard Internet protocols, so they can be in the same room or in different countries. ZEO, in fact, can distribute a Zope site all over the world. In this chapter we'll explore some interesting ways you can distribute your ZEO clients.

When you should use ZEO

ZEO serves many hits in a fail−safe way. If your site does not get millions of hits, then you probably don't need ZEO. There is no hard−and−fast rule about when you should and should not use ZEO, but for the most part you should not need to run ZEO unless:

- Your site is getting too many hits for your computer to handle them quickly. Zope is a high−performance system, and one Zope can handle millions of hits per day (depending on your hardware, of course). If you need to serve more hits than that, then you should use ZEO.
- Your site is very critical and requires constant, 24/7 uptime. In this case, ZEO will allow you to have multiple fail−over servers.
- You want to distribute your site globally to many different mirror ZEO clients.
- You want to debug one ZEO client while others are still serving requests. This is a very advanced technique for Python developers and is not covered in this book.

All of these cases are fairly advanced, high−end uses of Zope. Installing, configuring, and maintaining systems like these requires advanced system administration knowledge and resources. Most Zope users will not need ZEO, or may not have the expertise necessary to maintain a distributed server system like ZEO. ZEO is fun, and can be very useful, but before jumping head−first and installing ZEO in your system you should weigh the extra administrative burden ZEO creates against the simplicity of running just a simple, stand−alone Zope.

Installing and Running ZEO

The most common ZEO setup is one ZEO server and multiple ZEO clients. Before installing and configuring ZEO though, consider the following issues:

- All of the ZEO clients and servers must run the same version of Zope. Make sure all of your computers use the latest version. This is necessary, or Zope may behave abnormally or not work at all.
- All of your ZEO clients must have the same third party Products installed and they must be the same version. This is necessary, or your third−party objects may behave abnormally or not work at all.
- If your Zope system requires access to external resources, like mail servers or relational databases, ensure that all of your ZEO clients have access to those resources.
- Slow or intermittent network connections between clients and server degrade the performance of your ZEO clients. Your ZEO clients should have a good connection to their server.

ZEO is not distributed with Zope, you must download it from the Products Section of Zope.org.

Installing ZEO requires a little bit of manual preparation. To install ZEO, download the *ZEO−1.0.tgz* from the Zope.org web site and place it in your Zope installation directory. Now, unpack the tarball. On Unix, this can be done with the following command:

```
$ tar −zxf ZEO−1.0.tgz
```

On Windows, you can unpack the archive with WinZip. Before installing ZEO, make sure you back up your Zope system first.

Now you should have a *ZEO−1.0* directory. Next, you have to copy some files into your Zope top level *lib/python* directory. This can be done on UNIX with:

```
$ cp −R ZEO−1.0/ZEO lib/python
```

If you're running windows, you can use the following DOS commands to copy your ZEO files:

```
C:\...Zope\>xcopy ZEO−1.0\* lib\python /S
```

Now, you have to create a special file in your Zope root directory called *custom_zodb.py*. In that file, put the following python code:

```
import ZEO.ClientStorage
Storage=ZEO.ClientStorage.ClientStorage(('localhost',7700))
```

This will configure your Zope to run as a ZEO client. If you pass ClientStorage a tuple, as this code does, the tuple must have two elements, a string which contains the address to the server, and the port that the server is listening on. In this example, we're going to show you how to run both the clients and the servers on the same machine, so the machine name is set to `localhost`.

Now, you have ZEO properly configured to run on one computer. Try it out by first starting the server. Go to your Zope top level directory in a terminal window or DOS box and type:

```
python lib/python/ZEO/start.py −p 7700
```

This will start the ZEO server listening on port 7700 on your computer. Now, in another window, start up Zope like you normally would, with the *z2.py* script:

```
$ python z2.py −D

−−−−−−
2000−10−04T20:43:11 INFO(0) client Trying to connect to server
−−−−−−
2000−10−04T20:43:11 INFO(0) ClientStorage Connected to storage
−−−−−−
2000−10−04T20:43:12 PROBLEM(100) ZServer Computing default pinky
−−−−−−
2000−10−04T20:43:12 INFO(0) ZServer Medusa (V1.19) started at Wed Oct  4 15:43:12 2000
        Hostname: pinky.zopezoo.org
        Port:8080
```

Notice how in the above example, Zope tells you *client Trying to connect to server* and then *ClientStorage Connected to storage*. This means your ZEO client has successfully connected to your ZEO server. Now, you can visit *http://localhost:8080/manage* (or whatever URL your ZEO client is listening on) and log into Zope as usual.

As you can see, everything looks the same. Go to the *Control Panel* and click on *Database Managment*. Here, you see that Zope is connected to a *ZEO Storage* and that its state is *connected*.

Running ZEO on one computer is a great way to familiarize yourself with ZEO and how it works. Running ZEO on one computer does not, however, improve the speed of your site, and in fact, it may slow it down just a little. To really get the speed benefits that ZEO provides, you need to run ZEO on several computers, which is explained in the next section.

How to Run ZEO on Many Computers

Setting up ZEO to run on multiple computers is very similar to running ZEO on one computer. There are generally two steps, the first step is to start the ZEO server, and the second step is to start one or more ZEO clients.

For example, let's say you have four computers. One computer named *zooserver* will be your ZEO server, and the other three computers, named *zeoclient1*, *zeoclient2* and *zeoclient3*, will be your ZEO clients.

The first step is to run the server on *zooserver*. To tell your ZEO server to listen on the tcp socket at port 9999 on the *zooserver* interface, run the server with the *start.py* script like this:

```
$ python lib/python/ZEO/start.py −p 9999 −h zooserver.zopezoo.org
```

This will start the ZEO server. Now, you can start up your clients by going to each client and configuring each of them with the following *custom_zodb.py*:

```
import ZEO.ClientStorage
Storage=ZEO.ClientStorage.ClientStorage(('zooserver.zopezoo.org',9999))
```

Now, you can start each client's z2.py script as shown in the previous section, *Installing and Running ZEO*. Notice how the host and port for each client is the same, this is so they all connect to the same server. By following this procedure for each of your three clients you will have three different Zope's all serving the same Zope site. You can verify this by going visiting port 8080 on all three of your ZEO client machines.

You probably want to run ZEO on more than one computer so that you can take advantage of the speed increase this gives you. Running more computers means that you can serve more hits per second than with just one computer. Distributing the load of your web site's visitors however does require a bit more elaboration in your system. The next section describes why, and how, you distribute the load of your visitors among many computers.

How to Distribute Load

In the previous example you have a ZEO server named *zooServer* and three ZEO clients named *zeoclient1*, *zeoclient2*, and *zeoclient3*. The three ZEO clients are connected to the ZEO server and each client is verified to work properly.

Now you have three computers that serve content to your users. The next problem is how to actually spread the incoming web requests evenly among the three ZEO clients. Your users only know about *www.zopezoo.org*, not *zeoclient1*, *zeoclient2* or *zeoclient3*. It would be a hassle to tell only some users to use *zeoclient1*, and others to use *zeoclient3*, and it wouldn't be very good use of your computing resources. You want to automate, or at least make very easy, the process of evenly distributing requests to your various ZEO clients.

There are a number of solutions to this problem, some easy, some advanced, and some expensive. The next section goes over the more common ways of spreading web requests around various computers using different kinds of technology, some of them based on freely−available or commercial software, and some of them based on special hardware.

User Chooses a Mirror

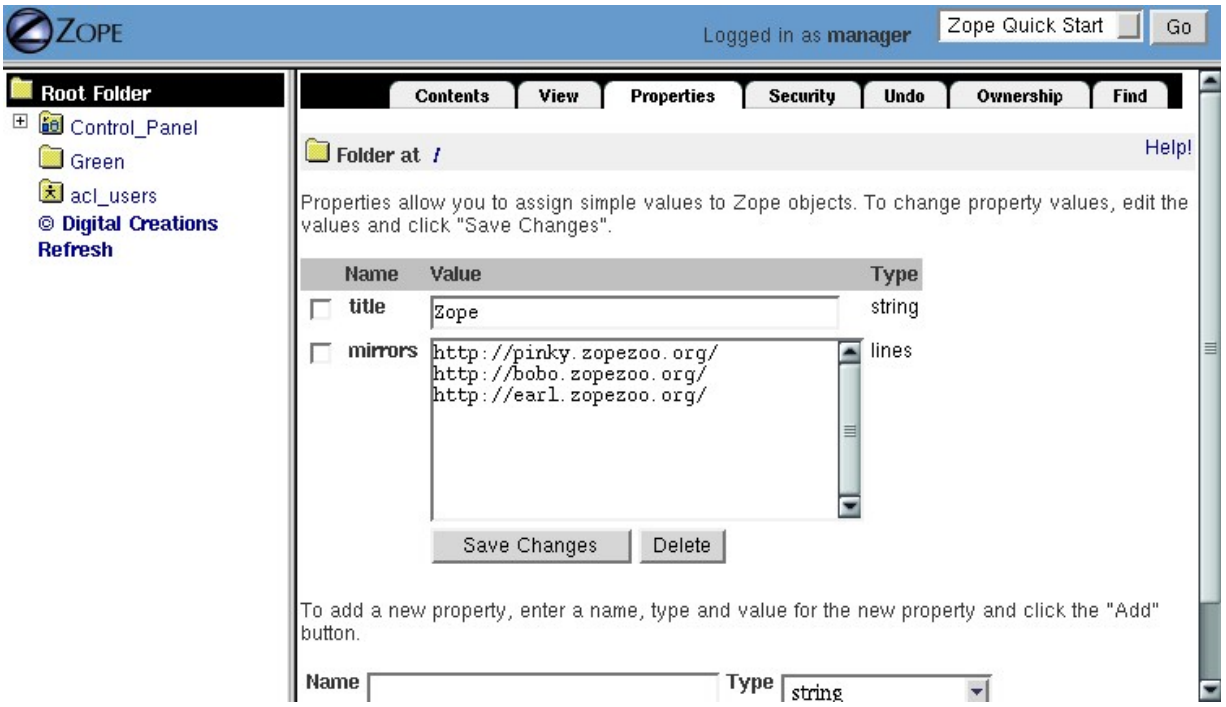
The easiest way to distribute requests across many web servers is to pick from a list of *mirrored sites*, each of which is a ZEO client. Using this method requires no extra software or hardware, it just requires the maintenance of a list of mirror servers. By presenting your users with a menu of mirrors, they can use to choose which server to use.

Note that this method of distributing requests is passive (you have no active control over which clients are used) and voluntary (your users need to make a voluntary choice to use another ZEO client). If your users do not use a mirror, then the requests will go to your ZEO client that serves *www.zopezoo.org*.

If you do not have any administrative control over your mirrors, then this can be a pretty easy solution. If your mirrors go off−line, your users can always choose to come back to the master site which you *do* have administrative control over and choose a different mirror.

On a global level, this method improves performance. Your users can choose to use a server that is geographically closer to them, which probably results in faster access. For example, if your main server was in Portland, Oregon on the west coast of the USA and you had users in London, England, they could choose your London mirror and their request would not have to go half−way across the world and back.

To use this method, create a property in your root folder of type *lines* named "mirror_servers". On each line of this property, put the URL to your various ZEO clients, as shown in Figure 11−2.



Folder at /

Properties allow you to assign simple values to Zope objects. To change property values, edit the values and click "Save Changes".

Name	Value	Type
title	Zope	string
mirrors	http://pinky.zopezoo.org/ http://bobo.zopezoo.org/ http://earl.zopezoo.org/	lines

To add a new property, enter a name, type and value for the new property and click the "Add" button.

Name Type string

Figure 11−2 Figure of property with URLs to mirrors

Now, add some simple DTML to your site to display a list of your mirrors:

```
<h2>Please choose from the following mirrors:
<ul>
  <dtml-in mirror_servers>
  <li><a href="&dtml-sequence-item;"><dtml-var
  sequence-item></a></li>
  </dtml-in>
</ul>
```

This DTML displays a list of all mirrors your users can choose from. When using this model, it is good to name your computers in ways that assist your users in their choice of mirror. For example, if you spread the load geographically, then choose names of countries for your computer names.

Alternatively, if you do not want users voluntarily choosing a mirror, you can have the *index_html* method of your www.zopezoo.org site issue HTTP redirects. For example, use the following code in your *www.zopezoo.org* site's *index_html* method:

```
<dtml-call expr="RESPONSE.redirect(_.whrandom.choice(mirror_servers))">
```

This code will redirect any visitors to *www.zopezoo.org* to a random mirror server.

Using Round−robin DNS to Distribute Load

The *Domain Name System*, or DNS, is the Internet mechanism that translates computer names (like "www.zope.org") into numeric addresses. This mechanism can map one name to many addresses.

The simplest method for load−balancing is to use round−robin DNS, as illustrated in Figure 11−3.

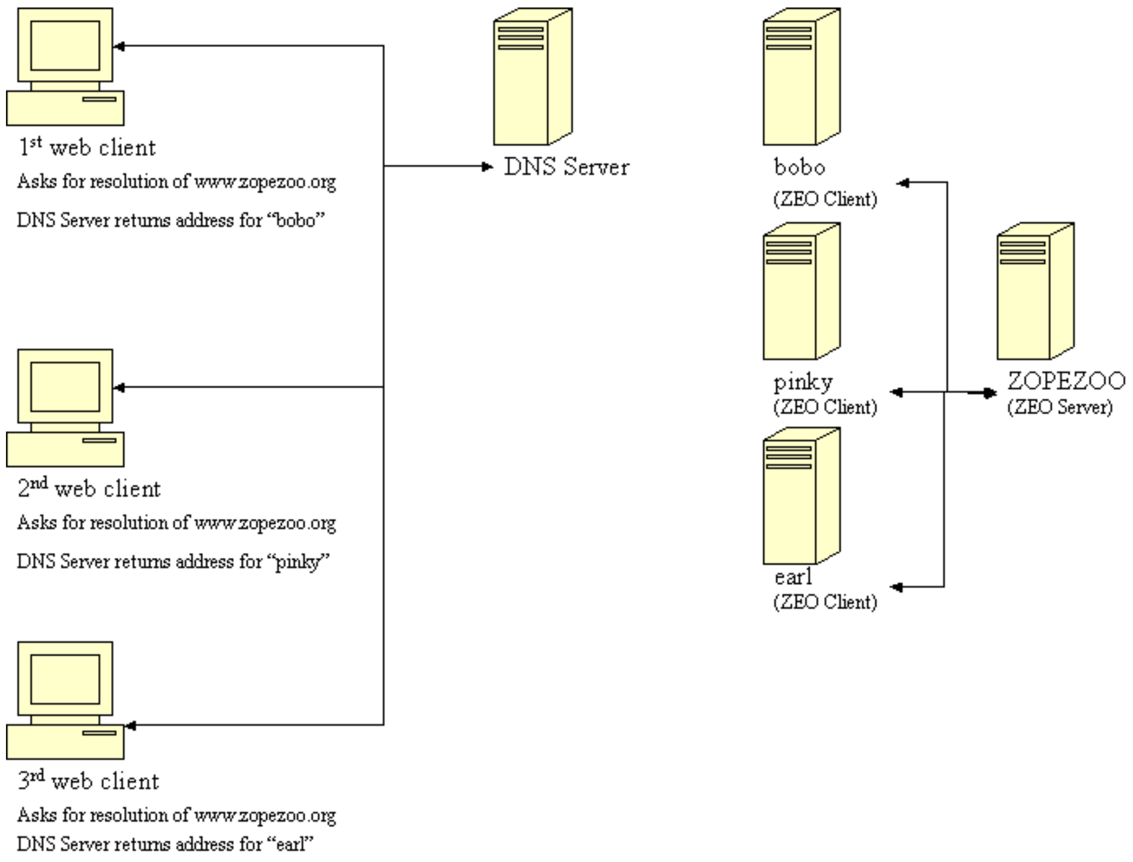

1st web client
Asks for resolution of www.zopezoo.org
DNS Server returns address for "bobo"
2nd web client
Asks for resolution of www.zopezoo.org
DNS Server returns address for "pinky"
3rd web client
Asks for resolution of www.zopezoo.org
DNS Server returns address for "earl"
DNS Server
bobo
(ZEO Client)
pinky
(ZEO Client)
earl
(ZEO Client)
ZOPEZOO
(ZEO Server)

Figure 11−3 Load balancing with round−robin DNS.

When *www.zopezoo.org* gets resolved, BIND answers with the address of either *zeoclient1*, *zeoclient2*, or *zeoclient3* − but in a rotated order every time. For example, one user may resolve *www.zopezoo.org* and get the address for *zeoclient1*, and another user may resolve *www.zopezoo.org* and get the address for *zeoclient2*. This way your users are spread over the various ZEO clients.

This not a perfect load balancing scheme, because DNS resolve information gets cached by the other nameservers on the net. Once a user has resolved *www.zopezoo.org* to a particular ZEO client, all subsequent requests for that user also go to the same ZEO client. The final result is generally alright, because the total sum of the requests are really spread over your various ZEO clients.

One down−side to this solution is that it can take from hours to days for name servers to refresh their cached copy of what they think the address of *www.zopezoo.org* is. If you are not responsible for the maintenance of your ZEO clients and one fails, then 1/Nth of your users (where N is the number of ZEO clients) will not be able to reach your site until their name server cache refreshes.

Configuring your DNS server to do round−robin name resolution is a pretty advanced technique that is not covered in this book. A good reference on how to do this can be found in the Apache Documentation.

Distributing the load with round−robin DNS is useful, and cheap, but not 100% effective. DNS servers can have strange caching policies, and you are relying on a particular quirk in the way DNS works to distribute the load. The next section describes a more complex, but much more powerful way of distributing load called *Layer 4 Switching*.

Using Layer 4 Switching to Distribute Load

Layer 4 switching lets one computer transparently hand requests to a farm of computers. This is a pretty advanced technique that is beyond the scope of this book, but it is worth pointing out several products that do Layer 4 switching for you.

Layer 4 switching involves a *switch* that, according to your preferences, chooses from a group of ZEO clients whenever a request comes in, as shown in Figure 11−4.

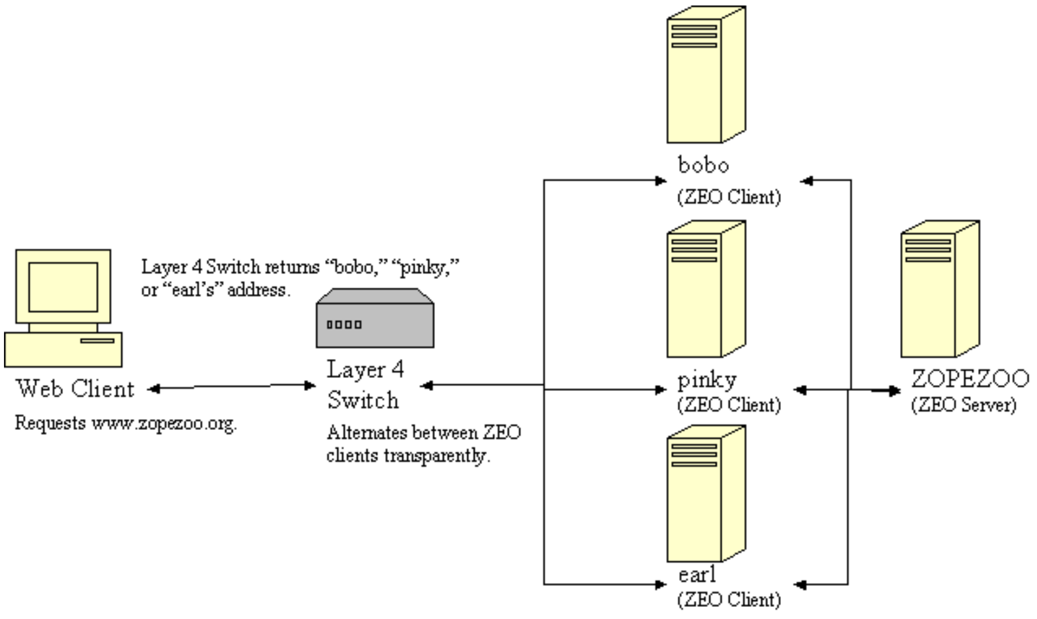

Layer 4 Switch returns "bobo," "pinky," or "earl's" address.
Web Client
Requests www.zopezoo.org.
Layer 4 Switch
Alternates between ZEO clients transparently.
bobo
(ZEO Client)
pinky
(ZEO Client)
earl
(ZEO Client)
ZOPEZOO
(ZEO Server)

Figure 11−4 Illustration of Layer 4 switching

There are hardware and software Layer 4 switches. There are a number of software solutions, but one in general that stands out is the *Linux Virtual Server* (LVS). This is an extension to the free Linux operating system that lets you turn a Linux computer into a Layer 4 switch. More information on the LVS can be found on its web site.

There are also a number of hardware solutions that claim higher performance than software based solutions like LVS. Cisco Systems has a hardware router called LocalDirector that works as a Layer 4 switch, and Alteon also makes a popular Layer 4 switch.

Dealing with a Single Point of Failure

Without ZEO, your entire Zope system is a single point of failure. ZEO allows you to spread that point of failure around to many different computers. If one of your ZEO clients fails, other clients can answer requests on the failed clients behalf.

Note that as of this writing, the single point of failure can't be *entirely* eliminated, because there is still one central storage server. The methods described in this section, however, do minimize the risks of failure by spreading most of Zope across many computers.

What this means is that, while this does remove a lot of risk away from your web servers as a single point of failure, it does not eliminate all risk because now the ZEO server is a single point of failure. There are several ways of dealing with this issue.

One popular method is to accept the single point of failure risk and mitigate that risk as much as possible by using very high−end, reliable equipment for your ZEO server, frequently backing up your data, and using inexpensive, off−the−shelf hardware for your ZEO clients. By investing the bulk of your infrastructure budget on making your ZEO server rock solid (redundant power supplies, RAID, and other fail−safe methods) you can be pretty well assured that your ZEO server will remain up, even if a handful of your inexpensive ZEO clients fail.

Some applications, however, require absolute 100% up−time. There is still a chance, with the solution described above, that your ZEO server will fail. If this happens, you want a backup ZEO server to jump in and take over for the failed server right away.

Like Layer 4 switching, there are a number of products, software and hardware, that help you mitigate this kind of risk. One popular software solution for linux is called fake. Fake is a Linux based utility that can make a backup computer take over for a failed primary computer by "faking out" network addresses. When used in conjunction with monitoring utilities like mon or heartbeat, fake can guarantee almost 100% up−time of your ZEO server and Layer 4 switches. Using `fake` in this way is beyond the scope of this book.

So far, we've explained these techniques for mitigating a single point of failure:

- Various tools (mirrors, round−robin DNS, Layer 4 switching) can be used to multiplex requests across multiple computers.
- ZEO can be used to distribute your database (ZEO server) to multiple ZEO clients.
- *fake*, and other tools can be used to provide redundant servers and Layer 4 switches.

The final piece of the puzzle is the ZEO server itself, and where it stores its information. If your primary ZEO server fails, how can your backup ZEO server ensure it has the most recent information that was contained in the primary server? As usual, there are several ways to solve this problem, and they are covered in the next section.

ZEO Server Details

Before explaining the details of how the ZEO server works, it is worth understanding some details about how Zope *storages* work in general.

Zope does not save any of its object or information directly to disk. Instead, Zope uses a *storage* component that takes care of all the details of where objects should be saved.

This is a very flexible model, because Zope no longer needs to be concerned about opening files, or reading and writing from databases, or sending data across a network (in the case of ZEO). Each particular storage takes care of that task on Zope's behalf.

For example, a plain, stand−alone Zope system can be illustrated in Figure 11−5.

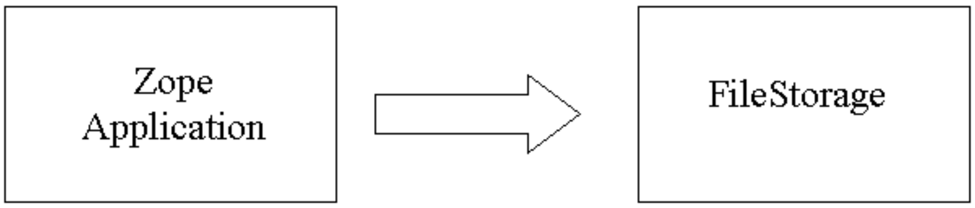


Figure 11−5 Zope connected to a filestorage

You can see there is one Zope application which plugs into a *FileStorage*. This storage, as its name implies, saves all of its information to a file on the computer's filesystem.

When using ZEO, you simple replace the FileStorage with a *ClientStorage*, as illustrated in Figure 11−6.

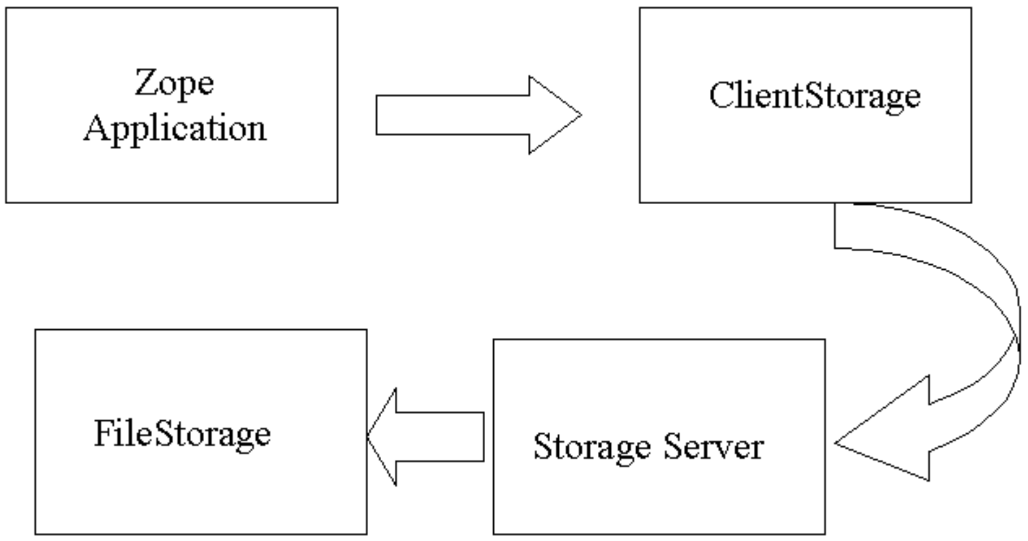


Figure 11−6 Zope with a Client Storage and Storage server

Instead of saving objects to a file, a ClientStorage sends objects over a network connection to a *Storage Server*. As you can see in the illustration, the Storage Server uses a FileStorage to save that information to a file on the ZEO server's filesystem.

Storages are interchangeable and easy to implement. Because of their interchangeable nature, ZEO Storage Servers can use ZEO ClientStorages to pass on object data to yet another ZEO Storage Server. This is illustrated in Figure 11−7.

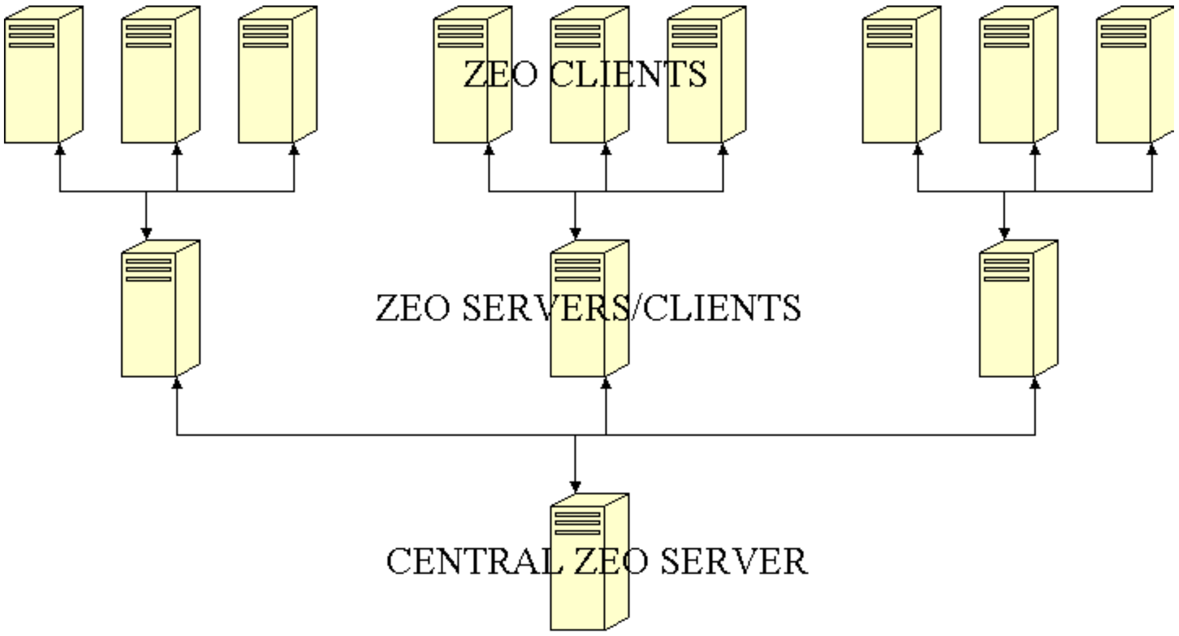


Figure 11−7 Multi−tiered ZEO system

Here, you can see a number of ZEO clients funnel down through three ZEO servers, which in turn act as ZEO clients themselves and funnel down into the final, central ZEO server than saves its information in a FileStorage. Now, that central ZEO server is the single point of failure in the system. If any of your other clients, or intermediate servers fail, the system will still continue to work, but if the central server fails, then you need an alternative.

Using `fake` you can have a back−up storage server strategy, but this method is not very well proven and hasn't been explored by the authors. In the future, ZEO will have a "multiple−server" feature, that allows a group of storage servers to act as a quorum, so if one or more storage servers fail, the remaining servers in the quorum can continue to serve objects.

There are a number of advantages to an approaches like these, especially if you are interested in creating a massively distributed network object database. Of course, with any system of advantages, there are some drawbacks as well, which are discussed in the next section.

ZEO Caveats

For the most part, running ZEO is exactly like running Zope by itself, but there are a few issues to keep in mind.

First, it takes longer for information to be written to the Zope object database. This does not slow down your ability to use Zope (because Zope does not block you during this write operation) but it does increase your chances of getting a *ConflictError*. Conflict errors happen when two ZEO clients try to write to the same object at the same time. One of the ZEO clients wins the conflict and continues on normally. The other ZEO client looses the conflict and has to try again.

Conflict errors should be as infrequent as possible because they could slow down your system. While it's normal to have a *few* conflict errors (due to the concurrent nature of Zope) it is abnormal to have *a lot* of conflict errors. The pathological case is when more than one ZEO client tries to write to the same object over and over again very quickly. In this case, there will be lots of conflict errors, and therefore lots of retries. If a ZEO client tries to write to the database three times and gets three conflict errors in a row, then the request is aborted and the data is not written.

Because ZEO takes longer to write this information, the chances of getting a ConflictError are higher than if you are not running ZEO. Because of this, ZEO is more *write sensitive* than running Zope without ZEO. You may have to keep this in mind when you are designing your network or application. As a rule of thumb, more and more frequent writes to the database increase your chances of getting a ConflictError. On the flip side, faster and more reliable network connections and computers lower your chances of getting a ConflictError. By taking these two factors into account, conflict errors can be mostly avoided.

Finally, as of this writing, there is no built in encryption or authentication between ZEO servers and clients. This means that you must be very careful about who you expose your ZEO servers to. If you leave your ZEO servers open to the whole Internet, then anyone can connect to your ZEO server and write data into your database, and that can be bad news.

This is not an unsolveable problem however, because you can use other tools, like firewalls, to protect your ZEO servers. If you are running a ZEO client/server connection over an unsecure network and you want guarantee that your information is kept private, you can use tools like OpenSSH and stunnel to set up secure, encrypted communication channels between your ZEO clients and servers. How these tools work and how to set them up is beyond the scope of this book, but both packages are adequately documented on their web sites. For more information on firewalls, with Linux in particular, we recommend the book "Linux Firewalls" by Robert Ziegler, which is published by New Riders.

Conclusion

In this chapter we looked at ZEO, and how ZEO can substantially increases the capacity of your website. In addition to running ZEO on one computer to get familiarized, we looked at running ZEO on many computers, and various techniques for spreading the load of your visitors among those many computers.

ZEO is not a magic bullet solution, and like other system designed to work with many computers, it adds another level of complexity to your web site. This complexity pays off however when you need to serve up lots of dynamic content to your audience.

Chapter 14: Extending Zope

You can extend Zope by creating your own types of objects that are customized to your applications needs. New kinds of objects are installed in Zope by *Products*. Products are extensions to Zope that Zope Corporation and many other third party developers create. There are hundreds of different Products and many serve very specific purposes. A complete library of Products is at the Download Section. of Zope.org.

Products can be developed two ways, *through the web* using ZClasses, and in the Python programming language. Products can even be a hybrid of both through the web products and Python code. This chapter discusses building new products through the web, a topic which you've already have some brief exposure to in Chapter 11, "Searching and Categorizing Content". Developing a Product entirely in Python product programming is the beyond its scope and you should visit Zope.org for specific Product developer documentation.

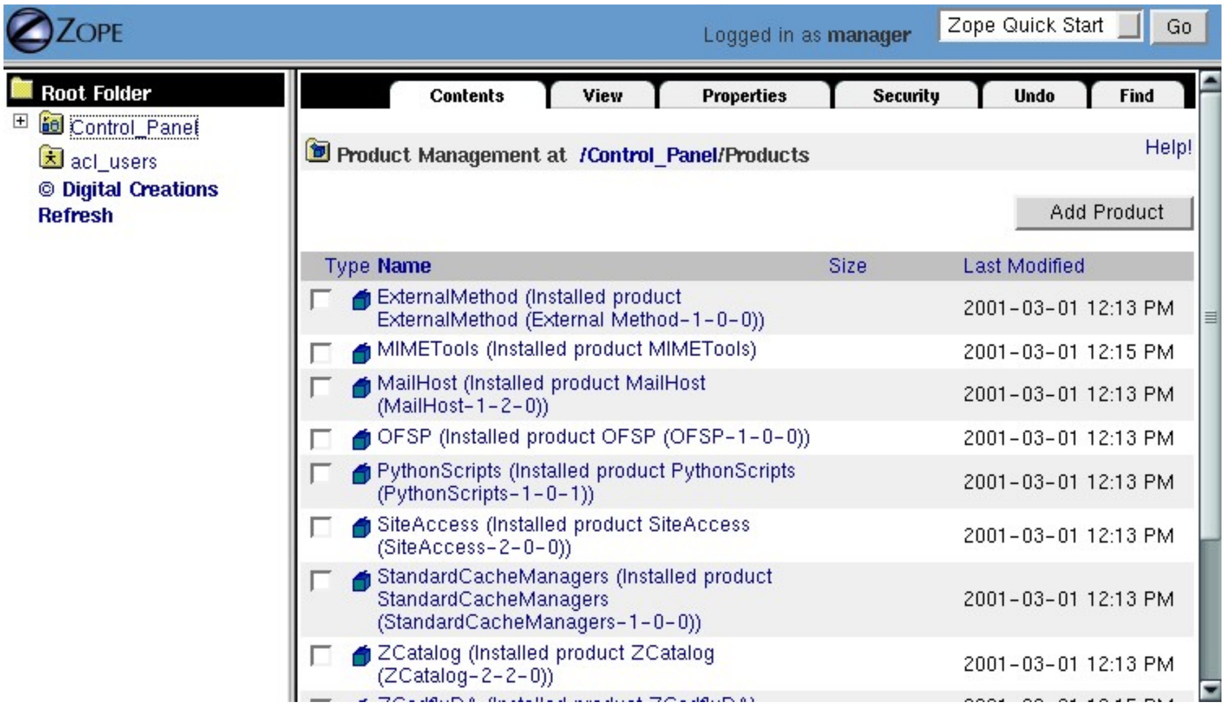
This chapter shows you how to:

- Create new Products in Zope
- Define ZClasses in Products
- Integrating Python with ZClasses
- Distribute Products to other Zope users

The first step in customizing Zope starts in the next section, where you learn how to create new Zope Products.

Creating Zope Products

Through the web Products are stored in the *Product Management* folder in the Control Panel. Click on the *Control_Panel* in the root folder and then click *Products*. You are now in the screen shown in Figure 12−1.



Product Management at /Control_Panel/Products

Type	Name	Size	Last Modified
	ExternalMethod (Installed product ExternalMethod (External Method-1-0-0))		2001-03-01 12:13 PM
	MIMETools (Installed product MIMETools)		2001-03-01 12:15 PM
	MailHost (Installed product MailHost (MailHost-1-2-0))		2001-03-01 12:13 PM
	OFSP (Installed product OFSP (OFSP-1-0-0))		2001-03-01 12:13 PM
	PythonScripts (Installed product PythonScripts (PythonScripts-1-0-1))		2001-03-01 12:13 PM
	SiteAccess (Installed product SiteAccess (SiteAccess-2-0-0))		2001-03-01 12:13 PM
	StandardCacheManagers (Installed product StandardCacheManagers (StandardCacheManagers-1-0-0))		2001-03-01 12:13 PM
	ZCatalog (Installed product ZCatalog (ZCatalog-2-2-0))		2001-03-01 12:13 PM

Figure 12−1 Installed Products

Each blue box represents an installed Product. From this screen, you can manage these Products. Some Products are built into Zope by default or have been installed by you or your administrator. These products have a *closed* box icon, as shown in Figure 12−1. Closed−box products cannot be managed through the web. You can get information about these products by clicking on them, but you cannot change them.

You can also create your own Products that you *can* manage through the web. Your products let you create new kinds of objects in Zope. These through the web managable product have open−box icons. If you followed the examples in Chapter 11, "Searching and Categorizing Content", then you have a *News* open−box product.

Why do you want to create products? For example, all of the various caretakers in the Zoo want an easy way to build simple on−line exhibits about the Zoo. The exhibits must all be in the same format and contain similar information structure, and each will be specific to a certain animal in the Zoo.

To accomplish this, you could build an exhibit for one animal, and then copy and paste it for each exhibit, but this would be a difficult and manual process. All of the information and properties would have to be changed for each new exhibit. Further, there may be thousands of exhibits.

To add to this problem, let's say you now want to have information on each exhibit that tells whether the animal is endangered or not. You would have to change each exhibit, one by one, to do this by using copy and paste. Clearly, copying and pasting does not scale up to a very large zoo, and could be very expensive.

You also need to ensure each exhibit is easy to manage. The caretakers of the individual exhibits should be the ones providing information, but none of the Zoo caretakers know much about Zope or how to create web sites and you certainly don't want to waste their time making them learn. You just want them to type some simple information into a form about their topic of interest, click submit, and walk away.

By creating a Zope product, you can acomplish these goals quickly and easily. You can create easy to manage objects that your caretakers can use. You can define exhibit templates that you can change once and effect all of the exhibits. You can do these things by creating Zope Products.

Creating A Simple Product

Using Products you can solve the exhibit creation and management problems. Let's begin with an example of how to create a simple product that will allow you to collect information about exhibits and create a customized exhibit. Later in the chapter you see more complex and powerful ways to use products.

The chief value of a Zope product is that it allows you to create objects in a central location and it gives you access to your objects through the product add list. This gives you the ability to build global services and make them available via a standard part of the Zope management interface. In other words a Product allows you to customize Zope.

Begin by going to the *Products* folder in the *Control Panel*. To create a new Product, click the *Add Product* button on the *Product Management* folder. This will take you to the Product add form. Enter the id "ZooExhibit" and click *Generate*. You will now see your new Product in the *Product Management* folder. It should be a blue box with an open lid. The open lid means you can click on the Product and manage it through the web.

Select the *ZooExhibit* Product. This will take you to the Product management screen.

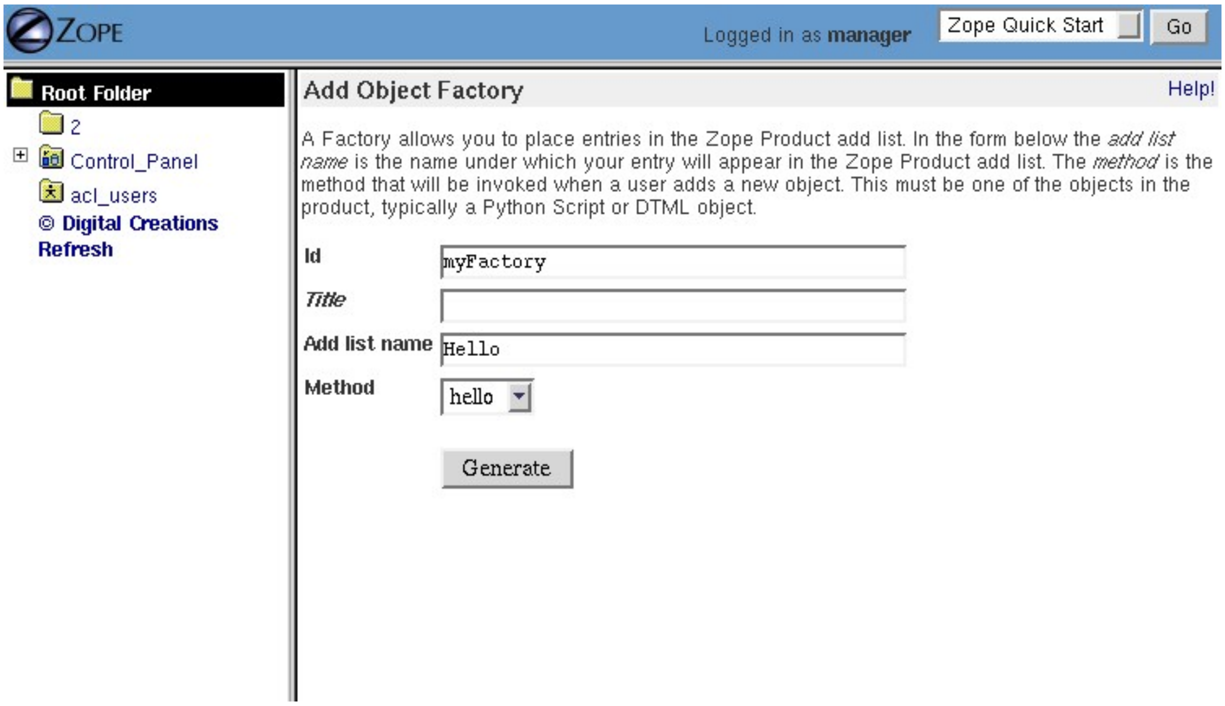
The management screen for a Product looks and acts just like a Folder except for a few differences:

1. There is a new view, called *Distribution*, all the way to the right. This gives you the ability to package and distribute your Product. This is discussed later.
2. If you select the add list, you will see some new types of objects you can add including *ZClass*, *Factory*, and *Permission*.
3. The folder with a question mark on it is the *ZooExhibit* Product's *Help Folder*. This folder can contain *Help Topics* that tell people how to use your Product.
4. There is also a new view *Define Permissions* that define the permissions associated with this Product. This is advanced and is not necessary for this example.

In the *Contents* View create a DTML Method named *hello* with these contents:

```
<dtml-var standard_html_header>

<h2>Hello from the Zoo Exhibit Product</h2>

<dtml-var standard_html_footer>
```

This method will allow you to test your product. Next create a Factory. Select *Zope Factory* from the product add list. You will be taken to a Factory add form as shown in Figure 12−2.



Add Object Factory

A Factory allows you to place entries in the Zope Product add list. In the form below the *add list name* is the name under which your entry will appear in the Zope Product add list. The *method* is the method that will be invoked when a user adds a new object. This must be one of the objects in the product, typically a Python Script or DTML object.

Id	myFactory
Title	
Add list name	Hello
Method	hello

Figure 12−2 Adding A Factory

Factories create a bridge from the product add list to your Product. Give your Factory an id of *myFactory*. In the *Add list name* field enter *Hello* and in the *Method* selection, choose *hello*. Now click *Generate*. Now click on the new Factory and change the *Permission* to *Add Document, Images, and Files* and click on *Save Changes*. This tells Zope that you must have the *Add Documents, Images, and Files* permission to use the Factory. Congratulations, you've just customized the Zope management interface. Go to the root folder and click the product add list. Notice that it now includes an entry named *Hello*. Choose *Hello* from the product add list. It calls your *hello* method.

One of the most common things to do with methods that you link to with Factories is to copy objects into the current Folder. In other words your methods can get access to the location from which they were called and can then perform operations on that Folder including copy objects into it. Just because you can do all kinds of crazy things with Factories and Products doesn't mean that you should. In general people expect that when they select something from the product add list that they will be taken to an add form where they specify the id of a new object. Then they expect that when they click *Add* that a new object with the id they specified will be created in their folder. Let's see how to fulfill these expectations.

First create a new Folder named *exhibitTemplate* in your Product. This will serve as a template for exhibits. Also in the Product folder create a DTML Method named *addForm*, and Python Script named *add*. These objects will create new exhibit instances. Now go back to your Factory and change it so that the *Add list name* is *Zoo Exhibit* and the method is *addForm*.

So what's going to happen is that when someone chooses *Zoo Exhibit* from the product add list, the *addForm* method will run. This method should collect information about the id and title of the exhibit. When the user clicks *Add* it should call the *add* script that will copy the *exhibitTemplate* folder into the calling folder and will rename it to have the specified id. The next step is to edit the *addForm* method to have these contents:

```
<dtml-var manage_page_header>

  <h2>Add a Zoo Exhibit</h2>

  <form action="add" method="post">
  id <input type="text" name="id"><br>
  title <input type="text" name="title"><br>
  <input type="submit" value=" Add ">
  </form>

<dtml-var manage_page_footer>
```

Admittedly this is a rather bleak add form. It doesn't collect much data and it doesn't tell the user what a Zoo Exhibit is and why they'd want to add one. When you create your own web applications you'll want to do better than this example.

Notice that this method doesn't include the standard HTML headers and footers. By convention Zope management screens don't use the same headers and footers that your site uses. Instead management screens use `manage_page_header` and `manage_page_footer`. The management view header and footer ensure that management views have a common look and feel.

Also notice that the action of the form is the *add* script. Now paste the following body into the *add* script:

```
## Script (Python) "add"
##parameters=id ,title, REQUEST=None
##
"""
Copy the exhibit template to the calling folder
"""

# Clone the template, giving it the new ID. This will be placed
# in the current context (the place the factory was called from).
exhibit=context.manage_clone(container.exhibitTemplate,id)

# Change the clone's title
exhibit.manage_changeProperties(title=title)

# If we were called through the web, redirect back to the context
if REQUEST is not None:
    try: u=context.DestinationURL()
    except: u=REQUEST['URL1']
    REQUEST.RESPONSE.redirect(u+'/manage_main?update_menu=1')
```

This script clones the *exhibitTemplate* and copies it to the current folder with the specified id. Then it changes the *title* property of the new exhibit. Finally it returns the current folder's main management screen by calling *manage_main*.

Congratulations, you've now extended Zope by creating a new product. You've created a way to copy objects into Zope via the product add list. However, this solution still suffers from some of the problems we discussed earlier in the chapter. Even though you can edit the exhibit template in a centralized place, it's still only a template. So if you add a new property to the template, it won't affect any of the existing exhibits. To change existing exhibits you'll have to modify each one manually.

ZClasses take you one step farther by allowing you to have one central template that defines a new type of object, and when you change that template, all of the objects of that type change along with it. This central template is called a ZClass. In the next section, we'll show you how to create ZClasses that define a new *Exhibit* ZClass.

Creating ZClasses

ZClasses are tools that help you build new types of objects in Zope by defining a *class*. A class is like a blueprint for objects. When defining a class, you are defining what an object will be like when it is created. A class can define methods, properties, and other attributes.

Objects that you create from a certain class are called *instances* of that class. For example, there is only one *Folder* class, but you many have many Folder instances in your application.

Instances have the same methods and properties as their class. If you change the class, then all of the instances reflect that change. Unlike the templates that you created in the last section, classes continue to exert control over instances. Keep in mind this only works one way, if you change an instance, no changes are made to the class or any other instances.

A good real world analogy to ZClasses are word processor templates. Most word processors come with a set of predefined templates that you can use to create a certain kind of document, like a resume. There may be hundreds of thousands of resumes in the world based on the Microsoft Word Resume template, but there is only one template. Like the Resume template is to all those resumes, a ZClass is a template for any number of similar Zope objects.

ZClasses are classes that you can build through the web using Zope's management interface. Classes can also be written in Python, but this is not covered in this book.

ZClasses can inherit attributes from other classes. Inheritance allows you to define a new class that is based on another class. For example, say you wanted to create a new kind of document object that had special properties you were interested in. Instead of building all of the functionality of a document from scratch, you can just *inherit* all of that functionality from the *DTML Document* class and add only the new information you are interested in.

Inheritance also lets you build generalization relationships between classes. For example, you could create a class called `Animal` that contains information that all animals have in general. Then, you could create *Reptile* and *Mammal* classes that both inherit from *Animal*. Taking it even further, you could create two additional classes *Lizard* and *Snake* that both inherit from *Reptile*, as shown in Figure 12−3.

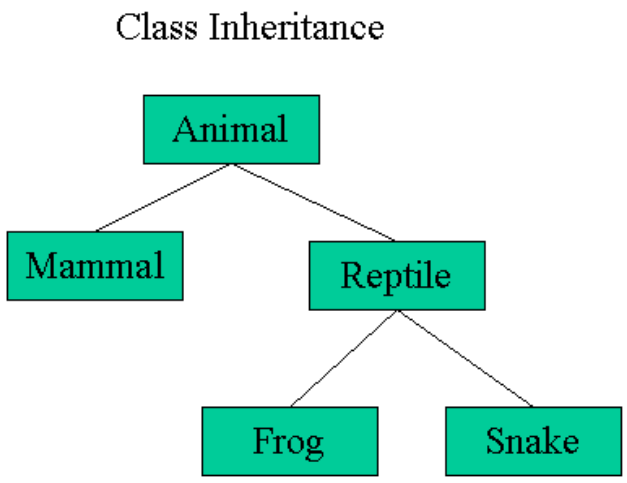


Figure 12−3 Example Class Inheritance

ZClasses can inherit from most of the objects you've used in this book. In addition, ZClasses can inherit from other ZClasses defined in the same Product. We will use this technique and others in this chapter.

Before going on with the next example, you should rename the existing *ZooExhibit* Product in your Zope Products folder to something else, like *ZooTemplate* so that it does not conflict with this example. Now, create a new Product in the Product folder called *ZooExhibit*.

Select *ZClass* from the add list of the *ZooExhibit* Contents view and go to the ZClass add form. This form is complex, and has lots of elements. We'll go through them one by one:

Id
: This is the name of the class to create. For this example, choose the name *ZooExhibit*.

Meta Type
: The Meta Type of an object is a name for the type of this object. This should be something short but descriptive about what the object does. For this example, choose the meta type "Zoo Exhibit".

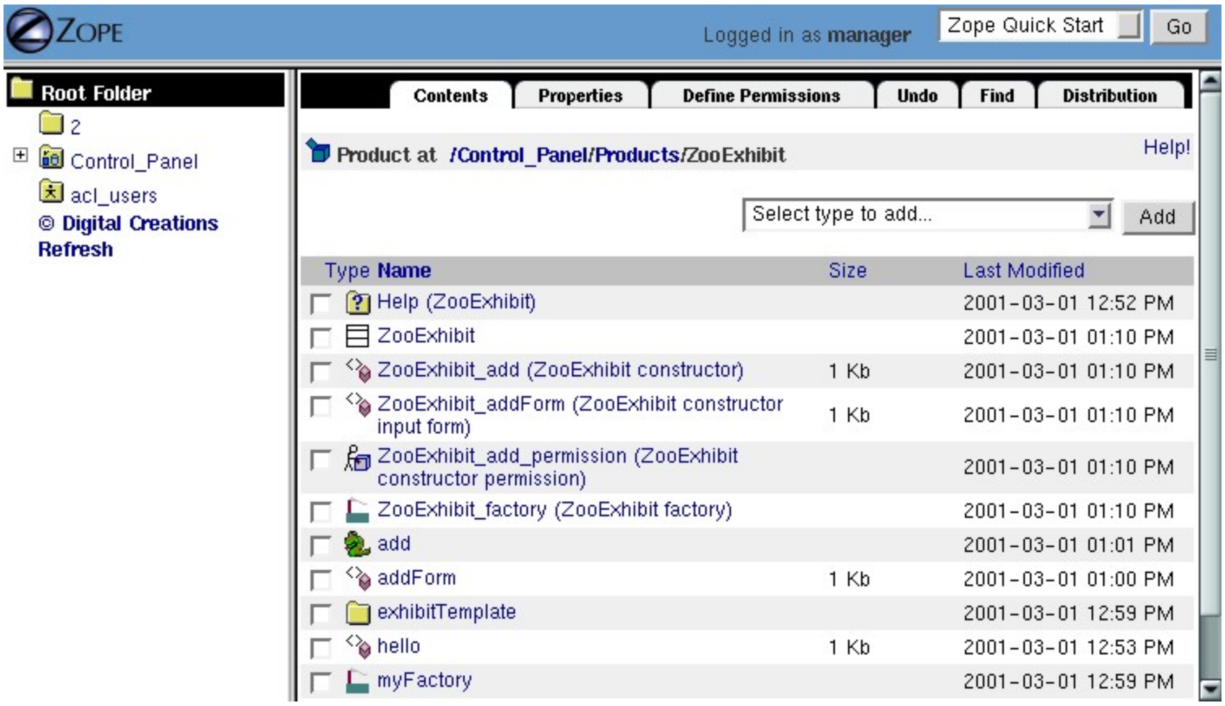
Base Classes
: Base classes define a sequence of classes that you want your class to inherit attributes from. Your new class can be thought of as *extending* or being *derived from* the functionality of your base classes. You can choose one or more classes from the list on the left, and click the −> button to put them in your base class list. The <− button removes any base classes you select on the right. For this example, don't select any base classes. Later in this chapter, we'll explain some of the more interesting base classes, like *ObjectManager*.

Create constructor objects?
: You usually want to leave this option checked unless you want to take care of creating form/action

constructor pairs and a Factory object yourself. If you want Zope to do this task for you, leave this checked. Checking this box means that this add form will create five objects, a Class, a Constructor Form, a Constructor Action, a Permission, and a Factory. For this example, leave this box checked.

Include standard Zope persistent object base classes?
: This option should be checked unless you don't want your object to be saved in the database. This is an advanced option and should only be used for Pluggable Brains. For this example, leave this box checked.

Now click *Add*. This will take you back to the *ZooExhibit* Product and you will see five new objects, as shown in Figure 12−4.



Product at /Control_Panel/Products/ZooExhibit

Type Name	Size	Last Modified
Help (ZooExhibit)		2001−03−01 12:52 PM
ZooExhibit		2001−03−01 01:10 PM
ZooExhibit_add (ZooExhibit constructor)	1 Kb	2001−03−01 01:10 PM
ZooExhibit_addForm (ZooExhibit constructor input form)	1 Kb	2001−03−01 01:10 PM
ZooExhibit_add_permission (ZooExhibit constructor permission)		2001−03−01 01:10 PM
ZooExhibit_factory (ZooExhibit factory)		2001−03−01 01:10 PM
add		2001−03−01 01:01 PM
addForm	1 Kb	2001−03−01 01:00 PM
exhibitTemplate		2001−03−01 12:59 PM
hello	1 Kb	2001−03−01 12:53 PM
myFactory		2001−03−01 12:59 PM

Figure 12−4 Product with a ZClass

The five objects Zope created are all automatically configured to work properly, you do not need to change them for now. Here is a brief description of each object that was created:

ZooExhibit
: This is the ZClass itself. It's icon is a white box with two horizontal lines in it. This is the traditional symbol for a *class*.

ZooExhibit_addForm
: This DTML Method is the constructor form for the ZClass. It is a simple form that accepts an id and title. You can customize this form to accept any kind of input your new object requires. The is very similar to the add form we created in the first example.

ZooExhibit_add
: This DTML Method gets called by the constructor form, *ZooExhibit_addForm*. This method actually creates your new object and sets its *id* and *title*. You can customize this form to do more advanced changes to your object based on input parameters from the *ZooExhibit_addForm*. This has the same functionality as the Python script we created in the previous example.

ZooExhibit_add_permission
: The curious looking stick−person carrying the blue box is a *Permission*. This defines a permission that you can associate with adding new *ZooExhibit* objects. This lets you protect the ability to add new Zoo exhibits. If you click on this Permission, you can see the name of this new permission is

"Add ZooExhibits".

ZooExhibit_factory
: The little factory with a smokestack icon is a *Factory* object. If you click on this object, you can change the text that shows up in the add list for this object in the *Add list name* box. The *Method* is the method that gets called when a user selects the *Add list name* from the add list. This is usually the constructor form for your object, in this case, *ZooExhibit_addForm*. You can associate the Permission the user must have to add this object, in this case, *ZooExhibit_add_permission*. You can also specify a regular Zope permission instead.

That's it, you've created your first ZClass. Click on the new ZClass and click on its *Basic* tab. The *Basic* view on your ZClass lets you change some of the information you specified on the ZClass add form. You cannot change the base classes of a ZClass. As you learned earlier in the chapter, these settings include:

meta−type
: The name of your ZClass as it appears in the product add list.

class id
: A unique identifier for your class. You should only change this if you want to use your class definition for existing instances of another ZClass. In this case you should copy the class id of the old class into your new class.

icon
: The path to your class's icon image. There is little reason to change this. If you want to change your class's icon, upload a new file with the *Browse* button.

At this point, you can start creating new instances of the *ZooExhibit* ZClass. First though, you probably want a common place where all exhibits are defined, so go to your root folder and select *Folder* from the add list and create a new folder with the id "Exhibits". Now, click on the *Exhibits* folder you just created and pull down the Add list. As you can see, *ZooExhibit* is now in the add list.

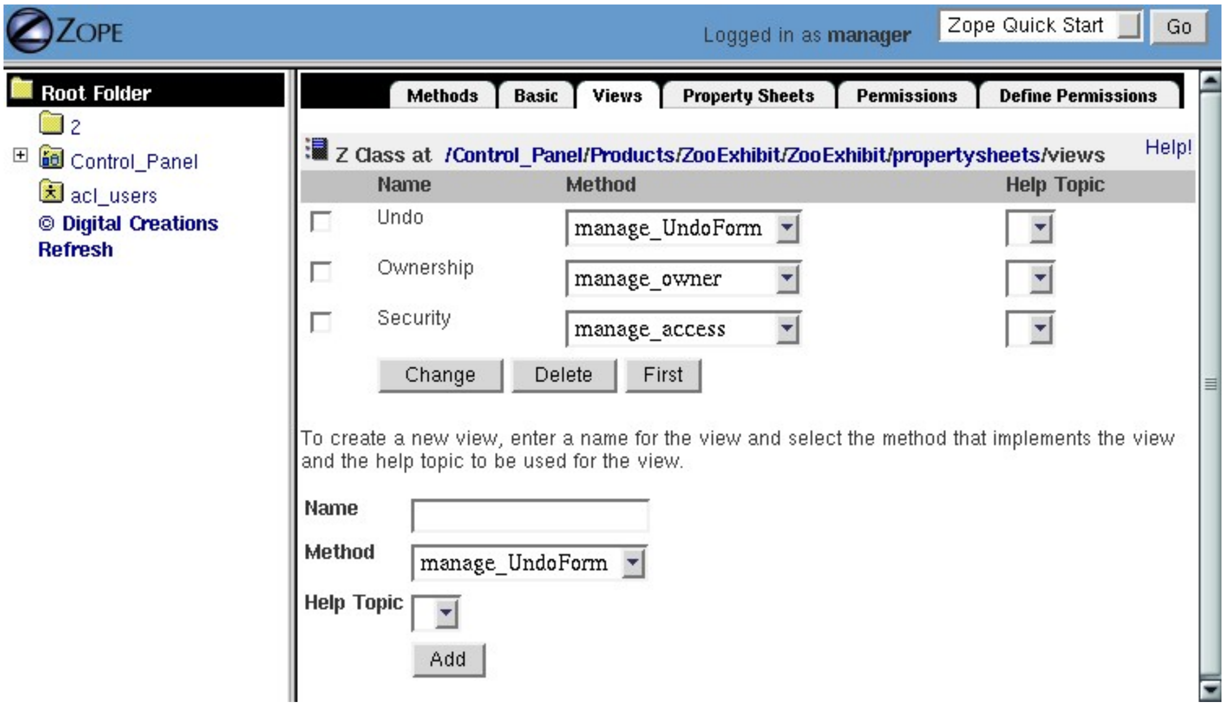
Go ahead and select *ZooExhibit* from the add list and create a new Exhibit with the id "FangedRabbits". After creating the new exhibit, select it by clicking on it.

As you can see your object already has three views, *Undo*, *Ownership*, and *Security*. You don't have to define these parts of your object, Zope does that for you. In the next section, we'll add some more views for you to edit your object.

Creating Views of Your ZClass

All Zope objects are divided into logical screens called *Views*. Views are used commonly when you work with Zope objects in the management interface, the tabbed screens on all Zope objects are views. Some views like *Undo*, are standard and come with Zope.

Views are defined on the *Views* view of a ZClass. Go to your *ZooExhibit* ZClass and click on the *Views* tab. The *Views* view looks like Figure 12−5.



Z Class at /Control_Panel/Products/ZooExhibit/ZooExhibit/propertysheets/views

	Name	Method	Help Topic
☐	Undo	manage_UndoForm	
☐	Ownership	manage_owner	
☐	Security	manage_access	

To create a new view, enter a name for the view and select the method that implements the view and the help topic to be used for the view.

Name	
Method	manage_UndoForm
Help Topic	

Figure 12−5 The Views view.

On this view you can see the three views that come automatically with your new object, *Undo*, *Ownership*, and *Security*. They are automatically configured for you as a convenience, since almost all objects have these interfaces, but you can change them or remove them from here if you really want to (you generally won't).

The table of views is broken into three columns, *Name*, *Method*, and *Help Topic*. The *Name* is the name of the view and is the label that gets drawn on the view's tab in the management interface. The *Method* is the method of the class or property sheet that gets called to render the view. The *Help Topic* is where you associate a *Help Topic* object with this view. Help Topics are explained more later.

Views also work with the security system to make sure users only see views on an object that they have permission to see. Security will be explained in detail a little further on, but it is good to know at this point that views now only divide an object management interfaces into logical chunks, but they also control who can see which view.

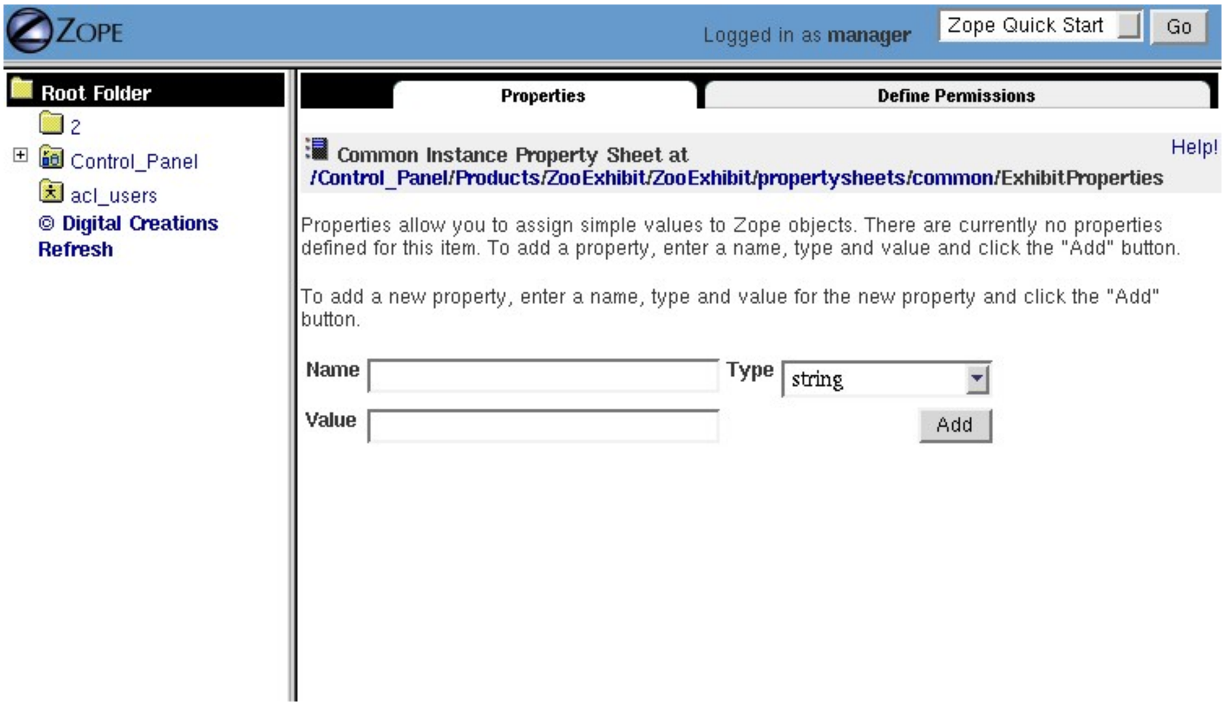
The *Method* column on the Methods view has select boxes that let you choose which method generates which view. The method associated with a view can be either an object in the *Methods* view, or a Property Sheet in the *Property Sheets* view.

Creating Properties on Your ZClass

Properties are collections of variables that your object uses to store information. A Zoo Exhibit object, for example, would need properties to contain information about the exhibit, like what animal is in the exhibit, a description, and who the caretakers are.

Properties for ZClasses work a little differently than properties on Zope objects. In ZClasses, Properties come in named groups called *Property Sheets*. A Property Sheet is a way of organizing a related set of properties together. Go to your *ZooExhibit* ZClass and click on the *Property Sheets* tab. To create a new sheet, click *Add Common Instance Property Sheet*. This will take you to the Property Sheet add form. Call your new Property Sheet "ExhibitProperties" and click *Add*.

Now you can see that your new sheet, *ExhibitProperties*, has been created in the *Property Sheets* view of your ZClass. Click on the new sheet to manage it, as shown in Figure 12−6.



Common Instance Property Sheet at /Control_Panel/Products/ZooExhibit/ZooExhibit/propertysheets/common/ExhibitProperties

Properties allow you to assign simple values to Zope objects. There are currently no properties defined for this item. To add a property, enter a name, type and value and click the "Add" button.

To add a new property, enter a name, type and value for the new property and click the "Add" button.

Figure 12−6 A Property Sheet

As you can see, this sheet looks very much like the *Properties* view on Zope objects. Here, you can create new properties on this sheet. Properties on Property Sheets are exactly like Properties on Zope objects, they have a name, a type, and a value.

Create three new properties on this sheet:

animal
: This property should be of type *string*. It will hold the name of the animal this exhibit features.

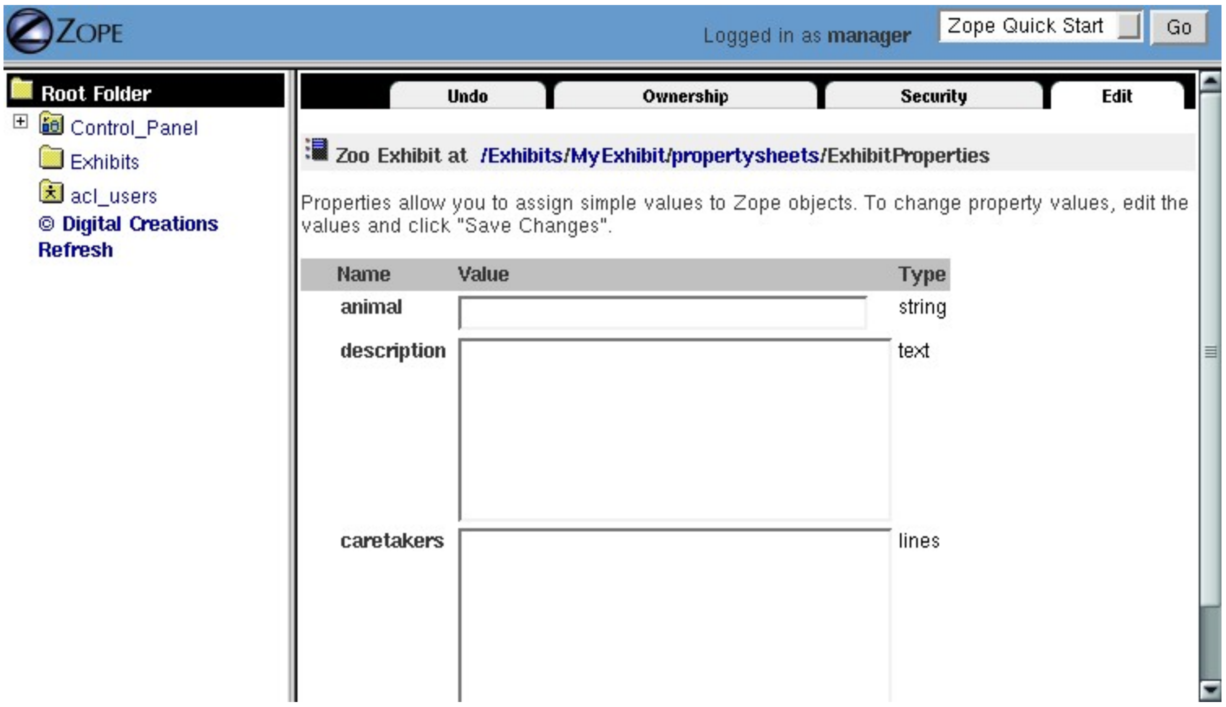
description
: This property should be of type *text*. It will hold the description of the exhibit.

caretakers
: This property should be of type *lines*. It will hold a list of names for the exhibit caretakers.

Property Sheets have two uses. As you've seen with this example, they are a tool for organizing related sets of properties about your objects, second to that, they are used to generate HTML forms and actions to edit those set of properties. The HTML edit forms are generated automatically for you, you only need to associate a view with a Property Sheet to see the sheet's edit form. For example, return to the ZooExhibit ZClass and click on the *Views* tab and create a new view with the name *Edit* and associate it with the method *propertysheets/ExhibitProperties/manage_edit*.

Since you can use Property Sheets to create editing screens you might want to create more than one Property Sheet for your class. By using more than one sheet you can control which properties are displayed together for editing purposes. You can also separate private from public properties on different sheets by associating them with different permissions.

Now, go back to your *Exhibits* folder and either look at an existing *ZooExhibit* instance or create a new one. As you can see, a new view called *Edit* has been added to your object, as shown in Figure Figure 12−7.



Zoo Exhibit at /Exhibits/MyExhibit/propertysheets/ExhibitProperties

Properties allow you to assign simple values to Zope objects. To change property values, edit the values and click "Save Changes".

Name	Value	Type
animal		string
description		text
caretakers		lines

Figure 12−7 A ZooExhibit Edit view

This edit form has been generated for you automatically. You only needed to create the Property Sheet, and then associate that sheet with a View. If you add another property to the *ExhibitProperties* Property Sheet, all of your instances will automatically get a new updated edit form, because when you change a ZClass, all of the instances of that class inherit the change.

It is important to understand that changes made to the class are reflected by all of the instances, but changes to an instance are *not* reflected in the class or in any other instance. For example, on the *Edit* view for your *ZooExhibit* instance (*not* the class), enter "Fanged Rabbit" for the *animal* property, the description "Fanged, carnivorous rabbits plagued early medieval knights. They are known for their sharp, pointy teeth." and two caretakers, "Tim" and "Somebody Else". Now click *Save Changes*.

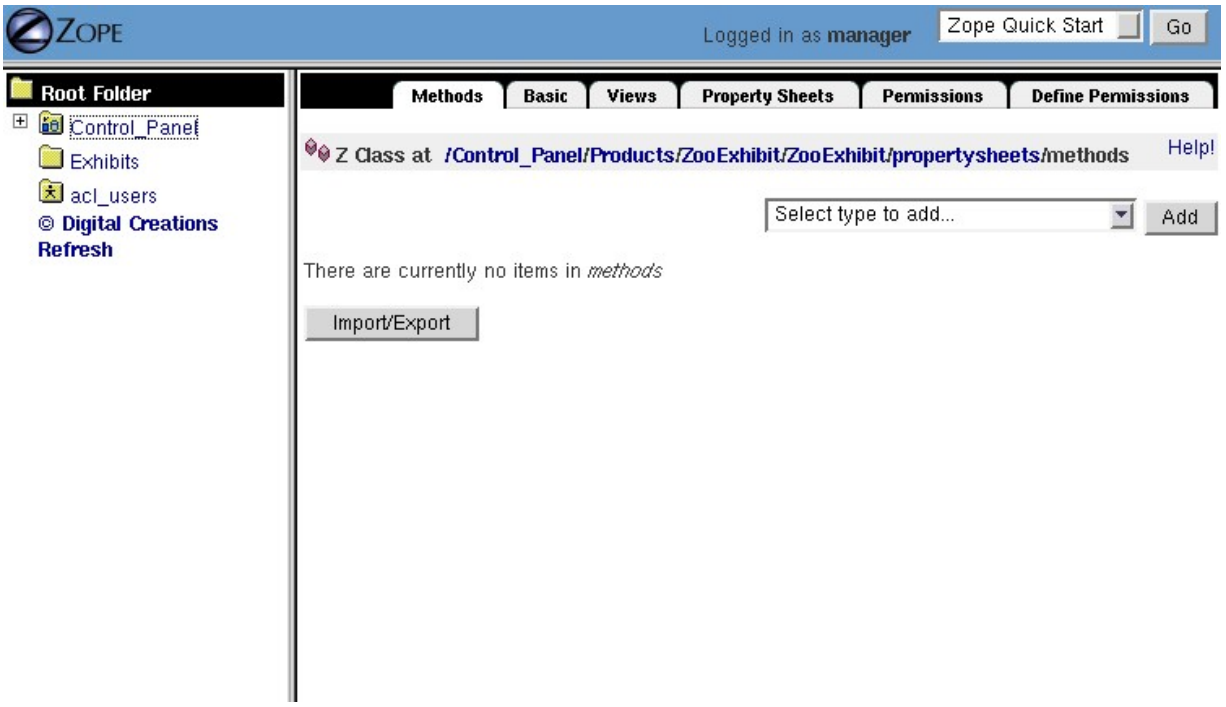
As you can see, your changes have obviously effected this instance, but what happened to the class? Go back to the *ZooExhibit* ZClass and look at the *ExhibitProperties* Property Sheet. Nothing has changed! Changes to instances have no effect on the class.

You can also provide default values for properties on a Property Sheet. You could, for example, enter the text "Describe your exhibit in this box" in the *description* property of the *ZooExhibit* ZClass. Now, go back to your *Exhibits* folder and create a *new* , *ZooExhibit* object and click on its *Edit* view. Here, you see that the value provided in the Property Sheet is the default value for the instance. Remember, if you change this instance, the default value of the property in the Property Sheet is *not* changed. Default values let you set up useful information in the ZClass for properties that can later be changed on an instance−by−instance basis.

You may want to go back to your ZClass and click on the *Views* tab and change the "Edit" view to be the first view by clicking the *First* button. Now, when you click on your instances, they will show the Edit view first.

Creating Methods on your ZClass

The *Methods* View of your ZClass lets you define the methods for the instances of your ZClass. Go to your *ZooExhibit* ZClass and click on the *Methods* tab. The *Methods* view looks like Figure 12−8.



Z Class at /Control_Panel/Products/ZooExhibit/ZooExhibit/propertysheets/methods

There are currently no items in *methods*

Figure 12−8 The Methods View

You can create any kind of Zope object on the *Methods* view, but generally only callable objects (DTML Methods and Scripts, for example) are added.

Methods are used for several purposes:

Presentation
: When you associate a view with a method, the method is called when a user selects that view on an instance. For example, if you had a DTML Method called *showAnimalImages*, and a view called *Images*, you could associate the *showAnimalImages* method with the *Images* view. Whenever anyone clicked on the *Images* view on an instance of your ZClass, the *showAnimalImages* method would get called.

Logic
: Methods are not necessarily associated with views. Methods are often created that define how you can work with your object.
For example, consider the *isHungry* method of the *ZooExhibit* ZClass defined later in this section. It does not define a view for a *ZooExhibit*, it just provide very specific information about the *ZooExhibit*. Methods in a ZClass can call each other just like any other Zope methods, so logic methods could be *used* from a presentation method, even though they don't *define* a view.

Shared Objects
: As was pointed out earlier, you can create any kind of object on the *Methods* view of a ZClass. All instances of your ZClass will *share* the objects on the Methods view. For example, if you create a *Z Gadfly Connection* in the Methods view of your ZClass, then all instances of that class will share the same Gadfly connection. Shared objects can be useful to your class's logic or presentation methods.

A good example of a presentation method is a DTML Method that displays a Zoo Exhibit to your web site viewers. This is often called the *public interface* to an object and is usually associated with the *View* view found on most Zope objects.

Create a new DTML Method on the *Methods* tab of your *ZooExhibit* ZClass called *index_html*. Like all objects named *index_html*, this will be the default representation for the object it is defined in, namely, instances of your ZClass. Put the following DTML in the *index_html* Method you just created:

```
<dtml-var standard_html_header>

  <h1><dtml-var animal></h1>

  <p><dtml-var description></p>

  <p>The <dtml-var animal> caretakers are:<br>
    <dtml-in caretakers>
      <dtml-var sequence-item><br>
    </dtml-in>
  </p>

<dtml-var standard_html_footer>
```

Now, you can visit one of your *ZooExhibit* instances directly through the web, for example, *http://www.zopezoo.org/Exhibits/FangedRabbits/* will show you the public interface for the Fanged Rabbit exhibit.

You can use Python−based or Perl−based Scripts, and even Z SQL Methods to implement logic. Your logic objects can call each other, and can be called from your presentation methods. To create the *isHungry* method, first create two new properties in the *ExhibitProperties* property sheet named "last_meal_time" that is of the type *date* and "isDangerous" that is of the type *boolean*. This adds two new fields to your Edit view where you can enter the last time the animal was fed and select whether or not the animal is dangerous.

Here is an example of an implementation of the *isHungry* method in Python:

```
## Script (Python) "isHungry"
##
"""
Returns true if the animal hasn't eaten in over 8 hours
"""
from DateTime import DateTime
if (DateTime().timeTime()
    − container.last_meal_time.timeTime() >  60 * 60 * 8):
    return 1
else:
    return 0
```

The `container` of this method refers to the ZClass instance. So you can use the `container` in a ZClass instance in the same way as you use `self` in normal Python methods.

You could call this method from your *index_html* display method using this snippet of DTML:

```
<dtml-if isHungry>
  <p><dtml-var animal> is hungry</p>
</dtml-if>
```

You can even call a number of logic methods from your display methods. For example, you could improve the hunger display like so:

```
<dtml-if isHungry>

  <p><dtml-var animal> is hungry.

  <dtml-if isDangerous>
```

```
            <a href="notify_hunger">Tell</a> an authorized
            caretaker.

          <dtml-else>

            <a href="feed">Feed</a> the <dtml-var animal>.

          </dtml-if>

          </p>

        </dtml-if>
```

Your display method now calls logic methods to decide what actions are appropriate and creates links to those actions. For more information on Properties, see Chapter 3, "Using Basic Zope Objects".

ObjectManager ZClasses

If you choose *ZClasses:ObjectManager* as a base class for your ZClass then instances of your class will be able to contain other Zope objects, just like Folders. Container classes are identical to other ZClasses with the exception that they have an addition view *Subobjects*.

From this view you can control what kinds of objects your instances can contain. For example if you created a FAQ container class, you might restrict it to holding Question and Answer objects. Select one or more meta−types from the select list and click the *Change* button. The *Objects should appear in folder lists* check box control whether or not instances of your container class are shown in the Navigator pane as expandable objects.

Container ZClasses can be very powerful. A very common pattern for web applications is to have two classes that work together. One class implements the basic behavior and hold data. The other class contains instances of the basic class and provides methods to organize and list the contained instances. You can model many problems this way, for example a ticket manager can contain problem tickets, or a document repository can contain documents, or an object router can contain routing rules, and so on. Typically the container class will provide methods to add, delete, and query or locate contained objects.

ZClass Security Controls

When building new types of objects, security can play an important role. For example, the following three Roles are needed in your Zoo:

Manager
: This role exists by default in Zope. This is you, and anyone else who you want to be able to completely manage your Zope system.

Caretaker
: After you create a *ZooExhibit* instance, you want users with the *Caretaker* role to be able to edit exhibits. Only users with this role should be able to see the *Edit* view of a *ZooExhibit* instance.

Anonymous
: This role exists by default in Zope. People with the *Anonymous* role should be able to view the exhibit, but not manage it or change it in any way.

As you learned in Chapter 7, "Users and Security", creating new roles is easy, but how can you control who can create and edit new *ZooExhibit* instances? To do this, you must define some security policies on the *ZooExhibit* ZClass that control access to the ZClass and its methods and property sheets.

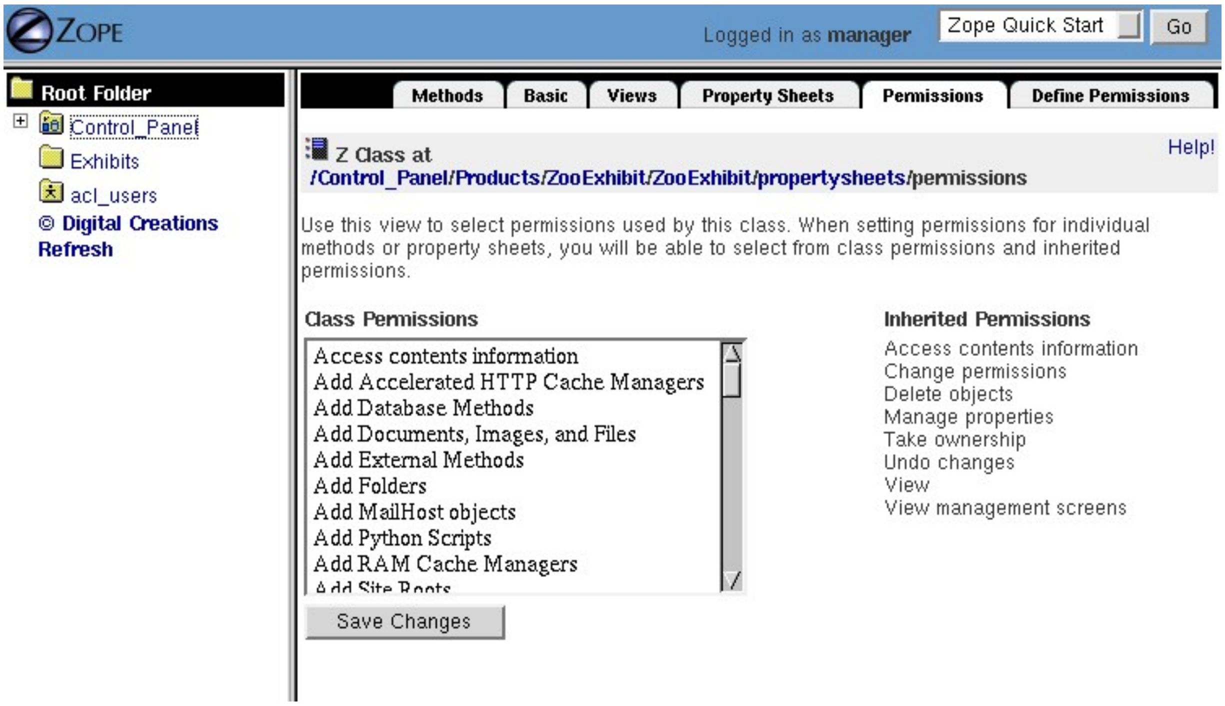
Controlling access to Methods and Property Sheets

By default, Zope tries to be sensible about ZClasses and security. You may, however, want to control access to instances of your ZClass in special ways.

For example, Zoo Caretakers are really only interested in seeing the *Edit* view (and perhaps the *Undo* view, which we'll show later), but definitely not the *Security* or *Ownership* views. You don't want Zoo caretakers changing the security settings on your Exhibits; you don't even want them to *see* those aspects of an Exhibit, you just want to give them the ability to edit an exhibit and nothing else.

To do this, you need to create a new *Zope Permission* object in the *ZooExhibit* Product (*not* the ZClass, permissions are defined in Products only). To do this, go to the *ZooExhibit* Product and select *Zope Permission* from the add list. Give the new permission the *Id* "edit_exhibit_permission" and the *Name* "Edit Zoo Exhibits" and click *Generate*.

Now, select your *ZooExhibit* ZClass, and click on the *Permissions* tab. This will take you to the *Permissions* view as shown in Figure Figure 12−9.



Z Class at /Control_Panel/Products/ZooExhibit/ZooExhibit/propertysheets/permissions

Use this view to select permissions used by this class. When setting permissions for individual methods or property sheets, you will be able to select from class permissions and inherited permissions.

Class Permissions

Access contents information
Add Accelerated HTTP Cache Managers
Add Database Methods
Add Documents, Images, and Files
Add External Methods
Add Folders
Add MailHost objects
Add Python Scripts
Add RAM Cache Managers
Add Site Roots

Inherited Permissions

Access contents information
Change permissions
Delete objects
Manage properties
Take ownership
Undo changes
View
View management screens

Figure 12−9 The Permissions view

This view shows you what permissions your ZClass uses and lets you choose additional permissions to use. On the right is a list of all of the default Zope permissions your ZClass inherits automatically. On the left is a multiple select box where you can add new permissions to your class. Select the *Edit Zoo Exhibits* permission in this box and click *Save Changes*. This tells your ZClass that it is interested in this permission as well as the permissions on the right.

Now, click on the *Property Sheets* tab and select the *ExhibitProperties* Property Sheet. Click on the *Define Permissions* tab.

You want to tell this Property Sheet that only users who have the *Edit Zoo Exhibits* permission you just created can manage the properties on the *ExhibitProperties* sheet. On this view, pull down the select box and choose *Edit Zoo Exhibits*. This will map the *Edit Zoo Exhibits* to the *Manage Properties* permission on the sheet. This list of permissions you can select from comes from the ZClass *Permissions* view you were just on, and because you selected the *Edit Zoo Exhibits* permission on that screen, it shows up on this list for you to select. Notice that all options default to *disabled* which means that the property sheet cannot be edited by anyone.

Now, you can go back to your *Exhibits* folder and select the *Security* view. Here, you can see your new Permission is on the left in the list of available permission. What you want to do now is create a new Role called *Caretaker* and map that new Role to the *Edit Zoo Exhibits* permission.

Now, users must have the *Caretaker* role in order to see or use the *Edit* view on any of your *ZooExhibit* instances.

Access to objects on your ZClass's *Methods* view are controlled in the same way.

Controlling Access to instances of Your ZClass

The previous section explained how you can control access to instances of your ZClass's Methods and Properties. Access control is controlling who can create new instances of your ZClass. As you saw earlier in the chapter, instances are created by Factories. Factories are associated with permissions. In the case of the Zoo Exhibit, the *Add Zoo Exhibits* permission controls the ability to create Zoo Exhibit instances.

Normally only Managers will have the *Add Zoo Exhibits* permission, so only Managers will be able to create new Zoo Exhibits. However, like all Zope permissions, you can change which roles have this permissions in different locations of your site. It's important to realize that this permission is controlled separately from the *Edit Zoo Exhibits* permission. This makes it possible to allow some people such as Caretakers to change, but not create Zoo Exhibits.

Providing Context−Sensitive Help for your ZClass

On the *View* screen of your ZClass, you can see that each view can be associated with a *Help Topic*. This allows you to provide a link to a different help topics depending on which view the user is looking at. For example, let's create a Help Topic for the *Edit* view of the *ZooExhibit* ZClass.

First, you need to create an actual help topic object. This is done by going to the *ZooExhibit* Product which contains the *ZooExhibit* ZClass, and clicking on the *Help* folder. The icon should look like a folder with a blue question mark on it.

Inside this special folder, pull down the add list and select *Help Topic*. Give this topic the id "ExhibitEditHelp" and the title "Help for Editing Exhibits" and click *Add*.

Now you will see the *Help* folder contains a new help topic object called *ExhibitEditHelp*. You can click on this object and edit it, it works just like a DTML Document. In this document, you should place the help information you want to show to your users:

```
<dtml−var standard_html_header>

  <h1>Help!</h1>

  <p>To edit an exhibit, click on either the <b>animal</b>,
  <b>description</b>, or <b>caretakers</b> boxes to edit
  them.</p>
```

```
        <dtml-var standard_html_footer>
```

Now that you have created the help topic, you need to associate with the *Edit* view of your ZClass. To do this, select the *ZooExhibit* ZClass and click on the *Views* tab. At the right, in the same row as the *Edit* view is defined, pull down the help select box and select *ExhibitEditHelp* and click *Change*. Now go to one of your ZooExhibit instances, the *Edit* view now has a *Help!* link that you can click to look at your Help Topic for this view.

In the next section, you'll see how ZClasses can be cobined with standard Python classes to extend their functionality into raw Python.

Using Python Base Classes

ZClasses give you a web managable interface to design new kinds of objects in Zope. In the beginning of this chapter, we showed you how you can select from a list of *base classes* to subclass your ZClass from. Most of these base classes are actually written in Python, and in this section you'll see how you can take your own Python classes and include them in that list so that your ZClasses can extend their methods.

Writing Python base classes is easy, but it involves a few installation details. To create a Python base class you need access to the filesystem. Create a directory inside your *lib/python/Products* directory named *AnimalBase*. In this directory create a file named *Animal.py* with these contents:

```
class Animal:
    """
    A base class for Animals
    """

    _hungry=0

    def eat(self, food, servings=1):
        """
        Eat food
        """
        self._hungry=0

    def sleep(self):
        """
        Sleep
        """
        self._hungry=1

    def hungry(self):
        """
        Is the Animal hungry?
        """
        return self._hungry
```

This class defines a couple related methods and one default attribute. Notice that like External Methods, the methods of this class can access private attributes.

Next you need to register your base class with Zope. Create an *__init__.py* file in the *AnimalBase* directory with these contents:

```
from Animal import Animal

def initialize(context):
    """
    Register base class
```

```
          """
          context.registerBaseClass(Animal)
```

Now you need to restart Zope in order for it find out about your base class. After Zope restarts you can verify that your base class has been registered in a couple different ways. First go to the Products Folder in the Control Panel and look for an *AnimalBase* package. You should see a closed box product. If you see broken box, it means that there is something wrong with your *AnimalBase* product.

Click on the *Traceback* view to see a Python traceback showing you what problem Zope ran into trying to register your base class. Once you resolve any problems that your base class might have you'll need to restart Zope again. Continue this process until Zope successfully loads your product. Now you can create a new ZClass and you should see *AnimalBase:Animal* as a choice in the base classes selection field.

To test your new base class create a ZClass that inherits from *AnimalBase:Animal*. Embellish you animal however you wish. Create a DTML Method named *care* with these contents:

```
<dtml-var standard_html_header>

<dtml-if give_food>
  <dtml-call expr="eat('cookie')">
</dtml-if>

<dtml-if give_sleep>
  <dtml-call sleep>
</dtml-if>

<dtml-if hungry>
  <p>I am hungry</p>
<dtml-else>
  <p>I am not hungry</p>
</dtml-if>

<form>
<input type="submit" value="Feed" name="give_food">
<input type="submit" value="Sleep" name="give_sleep">
</form>

<dtml-var standard_html_footer>
```

Now create an instance of your animal class and test out its *care* method. The care method lets you feed your animal and give it sleep by calling methods defined in its Python base class. Also notice how after feeding your animal is not hungry, but if you give it a nap it wakes up hungry.

As you can see, creating your own Products and ZClasses is an involved process, but simple to understand once you grasp the basics. With ZClasses alone, you can create some pretty complex web applications right in your web browser.

In the next section, you'll see how to create a *distribution* of your Product, so that you can share it with others or deliver it to a customer.

Distributing Products

Now you have created your own Product that lets you create any number of exhibits in Zope. Suppose you have a buddy at another Zoo who is impressed by your new online exhibit system, and wants to get a similar system for his Zoo.

Perhaps you even belong to the Zoo keeper's Association of America and you want to be able to give your product to anyone interested in an exhibit system similar to yours. Zope lets you distribute your Products as one, easy to transport package that other users can download from you and install in their Zope system.

To distribute your Product, click on the *ZooExhibit* Product and select the *Distribution* tab. This will take you to the *Distribution* view.

The form on this view lets you control the distribution you want to create. The *Version* box lets you specify the version for your Product distribution. For every distribution you make, Zope will increment this number for you, but you may want to specify it yourself. Just leave it at the default of "1.0" unless you want to change it.

The next two radio buttons let you select whether or not you want others to be able to customize or redistribute your Product. If you want them to be able to customize or redistribute your Product with no restrictions, select the *Allow Redistribution* button. If you want to disallow their ability to redistribute your Product, select the *Disallow redistribution and allow the user to configure only the selected objects:* button. If you disallow redistribution, you can choose on an object by object basis what your users can customize in your Product. If you don't want them to be able to change anything, then don't select any of the items in this list. If you want them to be able to change the *ZooExhibit* ZClass, then select only that ZClass. If you want them to be able to change everything (but still not be able to redistribute your Product) then select all the objects in this list.

Now, you can create a distribution of your Product by clicking *Create a distribution archive*. Zope will now automatically generate a file called *ZooExhibit−1.0.tar.gz*. This Product can be installed in any Zope just like any other Product, by unpacking it into the root directory of your Zope installation.

Don't forget that when you distribute your Product you'll also need to include any files such as External Method files and Python base classes that your class relies on. This requirement makes distribution more difficult and for this reason folks sometimes try to avoid relying on Python files when creating through the web Products for distribution.

Appendix A: DTML Reference

DTML is the *Document Template Markup Language*, a handy presentation and templating language that comes with Zope. This Appendix is a reference to all of DTMLs markup tags and how they work.

call: Call a method

The `call` tag lets you call a method without inserting the results into the DTML output.

Syntax

`call` tag syntax:

```
<dtml-call Variable|expr="Expression">
```

If the call tag uses a variable, the methods arguments are passed automatically by DTML just as with the `var` tag. If the method is specified in a expression, then you must pass the arguments yourself.

Examples

Calling by variable name:

```
<dtml-call UpdateInfo>
```

This calls the `UpdateInfo` object automatically passing arguments.

Calling by expression:

```
<dtml-call expr="RESPONSE.setHeader('content-type', 'text/plain')">
```

See Also

var tag

comment: Comments DTML

The comment tag lets you document your DTML with comments. You can also use it to temporarily disable DTML tags by commenting them out.

Syntax

`comment` tag syntax:

```
<dtml-comment>
</dtml-comment>
```

The `comment` tag is a block tag. The contents of the block are not executed, nor are they inserted into the DTML output.

Examples

Documenting DTML:

```
<dtml-comment>
  This content is not executed and does not appear in the
  output.
</dtml-comment>
```

Commenting out DTML:

```
<dtml-comment>
  This DTML is disabled and will not be executed.
  <dtml-call someMethod>
</dtml-comment>
```

functions: DTML Functions

DTML utility functions provide some Python built−in functions and some DTML−specific functions.

Functions

abs(number)
: Return the absolute value of a number. The argument may be a plain or long integer or a floating point number. If the argument is a complex number, its magnitude is returned.

chr(integer)
: Return a string of one character whose ASCII code is the integer, e.g., `chr(97)` returns the string `a`. This is the inverse of ord(). The argument must be in the range 0 to 255, inclusive; `ValueError` will be raised if the integer is outside that range.

DateTime()
: Returns a Zope `DateTime` object given constructor arguments. See the DateTime API reference for more information on constructor arguments.

divmod(number, number)
: Take two numbers as arguments and return a pair of numbers consisting of their quotient and remainder when using long division. With mixed operand types, the rules for binary arithmetic operators apply. For plain and long integers, the result is the same as `(a / b, a % b)`. For floating point numbers the result is `(q, a % b)`, where *q* is usually `math.floor(a / b)` but may be 1 less than that. In any case 'q * b + a % b' is very close to *a*, if `a % b` is non−zero it has the same sign as *b*, and `0 <= abs(a % b) < abs(b)`.

float(number)
: Convert a string or a number to floating point. If the argument is a string, it must contain a possibly signed decimal or floating point number, possibly embedded in whitespace; this behaves identical to `string.atof(number)`. Otherwise, the argument may be a plain or long integer or a floating point number, and a floating point number with the same value (within Python's floating point precision) is returned.

getattr(object, string)
: Return the value of the named attributed of object. name must be a string. If the string is the name of one of the object's attributes, the result is the value of that attribute. For example, `getattr(x, "foobar")` is equivalent to `x.foobar`. If the named attribute does not exist, default is returned if provided, otherwise `AttributeError` is raised.

getitem(variable, render=0)
: Returns the value of a DTML variable. If `render` is true, the variable is rendered. See the `render` function.

hasattr(object, string)
: The arguments are an object and a string. The result is 1 if the string is the name of one of the object's attributes, 0 if not. (This is implemented by calling getattr(object, name) and seeing whether it raises an exception or not.)

hash(object)
: Return the hash value of the object (if it has one). Hash values are integers. They are used to quickly compare dictionary keys during a dictionary lookup. Numeric values that compare equal have the same hash value (even if they are of different types, e.g. 1 and 1.0).

has_key(variable)
: Returns true if the DTML namespace contains the named variable.

hex(integer)
: Convert an integer number (of any size) to a hexadecimal string. The result is a valid Python expression. Note: this always yields an unsigned literal, e.g. on a 32−bit machine, `hex(−1)` yields `0xffffffff`. When evaluated on a machine with the same word size, this literal is evaluated as −1; at a different word size, it may turn up as a large positive number or raise an `OverflowError` exception.

int(number)
: Convert a string or number to a plain integer. If the argument is a string, it must contain a possibly signed decimal number representable as a Python integer, possibly embedded in whitespace; this behaves identical to 'string.atoi(number[, radix]'). The `radix` parameter gives the base for the conversion and may be any integer in the range 2 to 36. If `radix` is specified and the number is not a string, `TypeError` is raised. Otherwise, the argument may be a plain or long integer or a floating point number. Conversion of floating point numbers to integers is defined by the C semantics; normally the conversion truncates towards zero.

len(sequence)
: Return the length (the number of items) of an object. The argument may be a sequence (string, tuple or list) or a mapping (dictionary).

max(s)
: With a single argument s, return the largest item of a non−empty sequence (e.g., a string, tuple or list). With more than one argument, return the largest of the arguments.

min(s)
: With a single argument s, return the smallest item of a non−empty sequence (e.g., a string, tuple or list). With more than one argument, return the smallest of the arguments.

namespace([name=value]...)
: Returns a new DTML namespace object. Keyword argument `name=value` pairs are pushed into the new namespace.

oct(integer)
: Convert an integer number (of any size) to an octal string. The result is a valid Python expression. Note: this always yields an unsigned literal, e.g. on a 32−bit machine, `oct(−1)` yields `037777777777`. When evaluated on a machine with the same word size, this literal is evaluated as −1; at a different word size, it may turn up as a large positive number or raise an OverflowError exception.

ord(character)
: Return the ASCII value of a string of one character. E.g., `ord("a")` returns the integer 97. This is the inverse of `chr()`.

pow(x, y [,z])
: Return *x* to the power *y*; if *z* is present, return *x* to the power *y*, modulo *z* (computed more efficiently than 'pow(x, y) % z'). The arguments must have numeric types. With mixed operand types, the rules for binary arithmetic operators apply. The effective operand type is also the type of the result; if the result is not expressible in this type, the function raises an exception; e.g., `pow(2, −1)` or `pow(2, 35000)` is not allowed.

range([start,] stop [,step])

This is a versatile function to create lists containing arithmetic progressions. The arguments must be plain integers. If the step argument is omitted, it defaults to 1. If the start argument is omitted, it defaults to 0. The full form returns a list of plain integers '[start, start + step, start + 2 * step, ...]'. If step is positive, the last element is the largest 'start + i *step' less than* stop*; if* step *is negative, the last element is the largest 'start + i* step' greater than *stop*. *step* must not be zero (or else `ValueError` is raised).

round(x [,n])

Return the floating point value *x* rounded to *n* digits after the decimal point. If n is omitted, it defaults to zero. The result is a floating point number. Values are rounded to the closest multiple of 10 to the power minus n; if two multiples are equally close, rounding is done away from 0 (so e.g. round(0.5) is 1.0 and round(−0.5) is −1.0).

render(object)

Render `object`. For DTML objects this evaluates the DTML code with the current namespace. For other objects, this is equivalent to `str(object)`.

reorder(s [,with] [,without])

Reorder the items in s according to the order given in `with` and without the items mentioned in `without`. Items from s not mentioned in with are removed. s, with, and without are all either sequences of strings or sequences of key−value tuples, with ordering done on the keys. This function is useful for constructing ordered select lists.

SecurityCalledByExecutable()

Return a true if the current object (e.g. DTML document or method) is being called by an executable (e.g. another DTML document or method, a script or a SQL method).

SecurityCheckPermission(permission, object)

Check whether the security context allows the given permission on the given object. For example, 'SecurityCheckPermission("Add Documents, Images, and Files", this())' would return true if the current user was authorized to create documents, images, and files in the current location.

SecurityGetUser()

Return the current user object. This is normally the same as the `REQUEST.AUTHENTICATED_USER` object. However, the `AUTHENTICATED_USER` object is insecure since it can be replaced.

SecurityValidate([object] [,parent] [,name] [,value])

Return true if the value is accessible to the current user. `object` is the object the value was accessed in, `parent` is the container of the value, and `name` is the named used to access the value (for example, if it was obtained via 'getattr'). You may omit some of the arguments, however it is best to provide all available arguments.

SecurityValidateValue(object)

Return true if the object is accessible to the current user. This function is the same as calling `SecurityValidate(None, None, None, object)`.

str(object)

Return a string containing a nicely printable representation of an object. For strings, this returns the string itself.

test(condition, result [,condition, result]... [,default])

Takes one or more condition, result pairs and returns the result of the first true condition. Only one result is returned, even if more than one condition is true. If no condition is true and a default is given, the default is returned. If no condition is true and there is no default, None is returned.

unichr(number)

Return a unicode string representing the value of number as a unicode character. This is the inverse of ord() for unicode characters.

unicode(string[, encoding[, errors]])

Decodes string using the codec for encoding. Error handling is done according to errors. The default behavior is to decode UTF−8 in strict mode, meaning that encoding errors raise ValueError.

Attributes

None
: The `None` object is equivalent to the Python built−in object `None`. This is usually used to represent a Null or false value.

See Also

`string` module

`random` module

`math` module

`sequence` module

Built−in Python Functions

if: Tests Conditions

The `if` tags allows you to test conditions and to take different actions depending on the conditions. The `if` tag mirrors Python's `if/elif/else` condition testing statements.

Syntax

If tag syntax:

```
<dtml−if ConditionVariable|expr="ConditionExpression">
[<dtml−elif ConditionVariable|expr="ConditionExpression">]
 ...
[<dtml−else>]
</dtml−if>
```

The `if` tag is a block tag. The `if` tag and optional `elif` tags take a condition variable name or a condition expression, but not both. If the condition name or expression evaluates to true then the `if` block is executed. True means not zero, an empty string or an empty list. If the condition variable is not found then the condition is considered false.

If the initial condition is false, each `elif` condition is tested in turn. If any `elif` condition is true, its block is executed. Finally the optional `else` block is executed if none of the `if` and `elif` conditions were true. Only one block will be executed.

Examples

Testing for a variable:

```
<dtml−if snake>
  The snake variable is true
</dtml−if>
```

Testing for expression conditions:

```
<dtml−if expr="num > 5">
```

```
  num is greater than five
<dtml-elif expr="num < 5">
  num is less than five
<dtml-else>
  num must be five
</dtml-if>
```

See Also

Python Tutorial: If Statements

in: Loops over sequences

The `in` tag gives you powerful controls for looping over sequences and performing batch processing.

Syntax

`in` tag syntax:

```
<dtml-in SequenceVariable|expr="SequenceExpression">
[<dtml-else>]
</dtml-in>
```

The `in` block is repeated once for each item in the sequence variable or sequence expression. The current item is pushed on to the DTML namespace during each executing of the `in` block.

If there are no items in the sequence variable or expression, the optional `else` block is executed.

Attributes

mapping
: Iterates over mapping objects rather than instances. This allows values of the mapping objects to be accessed as DTML variables.

reverse
: Reverses the sequence.

sort=string
: Sorts the sequence by the given attribute name.

start=int
: The number of the first item to be shown, where items are numbered from 1.

end=int
: The number of the last item to be shown, where items are numbered from 1.

size=int
: The size of the batch.

skip_unauthorized
: Don't raise an exception if an unauthorized item is encountered.

orphan=int
: The desired minimum batch size. This controls how sequences are split into batches. If a batch smaller than the orphan size would occur, then no split is performed, and a batch larger than the batch size results.
: For example, if the sequence size is 12, the batch size is 10 the orphan size is 3, then the result is one batch with all 12 items since splitting the items into two batches would result in a batch smaller than the orphan size.

The default value is 0.

overlap=int
: The number of items to overlap between batches. The default is no overlap.

previous
: Iterates once if there is a previous batch. Sets batch variables for previous sequence.

next
: Iterates once if there is a next batch. Sets batch variables for the next sequence.

prefix=string
: Provide versions of the tag variables that start with this prefix instead of "sequence", and that use underscores (_) instead of hyphens (−). The prefix must start with a letter and contain only alphanumeric characters and underscores (_).

sort_expr=expression
: Sorts the sequence by an attribute named by the value of the expression. This allows you to sort on different attributes.

reverse_expr=expression
: Reverses the sequence if the expression evaluates to true. This allows you to selectively reverse the sequence.

Tag Variables

Current Item Variables

These variables describe the current item.

sequence−item
: The current item.

sequence−key
: The current key. When looping over tuples of the form `(key,value)`, the `in` tag interprets them as `(sequence−key, sequence−item)`.

sequence−index
: The index starting with 0 of the current item.

sequence−number
: The index starting with 1 of the current item.

sequence−roman
: The index in lowercase Roman numerals of the current item.

sequence−Roman
: The index in uppercase Roman numerals of the current item.

sequence−letter
: The index in lowercase letters of the current item.

sequence−Letter
: The index in uppercase letters of the current item.

sequence−start
: True if the current item is the first item.

sequence−end
: True if the current item is the last item.

sequence−even
: True if the index of the current item is even.

sequence−odd
: True if the index of the current item is odd.

sequence−length
: The length of the sequence.

sequence−var−variable
: A variable in the current item. For example, `sequence−var−title` is the `title` variable of the current item. Normally you can access these variables directly since the current item is pushed on the DTML namespace. However these variables can be useful when displaying previous and next batch information.

sequence−index−variable
: The index of a variable of the current item.

Summary Variables

These variable summarize information about numeric item variables. To use these variable you must loop over objects (like database query results) that have numeric variables.

total−variable
: The total of all occurrences of an item variable.

count−variable
: The number of occurrences of an item variable.

min−variable
: The minimum value of an item variable.

max−variable
: The maximum value of an item variable.

mean−variable
: The mean value of an item variable.

variance−variable
: The variance of an item variable with count−1 degrees of freedom.

variance−n−variable
: The variance of an item variable with n degrees of freedom.

standard−deviation−variable
: The standard−deviation of an item variable with count−1 degrees of freedom.

standard−deviation−n−variable
: The standard−deviation of an item variable with n degrees of freedom.

Grouping Variables

These variables allow you to track changes in current item variables.

first−variable
: True if the current item is the first with a particular value for a variable.

last−variable
: True if the current item is the last with a particular value for a variable.

Batch Variables

sequence−query
: The query string with the `start` variable removed. You can use this variable to construct links to next and previous batches.

sequence−step−size
: The batch size.

previous−sequence
: True if the current batch is not the first one. Note, this variable is only true for the first loop iteration.

previous−sequence−start−index
: The starting index of the previous batch.

previous−sequence−start−number
: The starting number of the previous batch. Note, this is the same as `previous−sequence−start−index` + 1.

previous−sequence−end−index
: The ending index of the previous batch.

previous−sequence−end−number
: The ending number of the previous batch. Note, this is the same as `previous−sequence−end−index` + 1.

previous−sequence−size
: The size of the previous batch.

previous−batches
: A sequence of mapping objects with information about all previous batches. Each mapping object has these keys `batch−start−index`, `batch−end−index`, and `batch−size`.

next−sequence
: True if the current batch is not the last batch. Note, this variable is only true for the last loop iteration.

next−sequence−start−index
: The starting index of the next sequence.

next−sequence−start−number
: The starting number of the next sequence. Note, this is the same as `next−sequence−start−index` + 1.

next−sequence−end−index
: The ending index of the next sequence.

next−sequence−end−number
: The ending number of the next sequence. Note, this is the same as `next−sequence−end−index` + 1.

next−sequence−size
: The size of the next index.

next−batches
: A sequence of mapping objects with information about all following batches. Each mapping object has these keys `batch−start−index`, `batch−end−index`, and `batch−size`.

Examples

Looping over sub−objects:

```
<dtml−in objectValues>
  title: <dtml−var title><br>
</dtml−in>
```

Looping over two sets of objects, using prefixes:

```
<dtml−let rows="(1,2,3)" cols="(4,5,6)">
  <dtml−in rows prefix="row">
    <dtml−in cols prefix="col">
      <dtml−var expr="row_item * col_item"><br>
      <dtml−if col_end>
        <dtml−var expr="col_total_item * row_mean_item">
      </dtml−if>
    </dtml−in>
  </dtml−in>
</dtml−let>
```

Looping over a list of `(key, value)` tuples:

```
<dtml−in objectItems>
```

```
  id: <dtml-var sequence-key>, title: <dtml-var title><br>
</dtml-in>
```

Creating alternate colored table cells:

```
<table>
<dtml-in objectValues>
<tr <dtml-if sequence-odd>bgcolor="#EEEEEE"
    <dtml-else>bgcolor="#FFFFFF"
    </dtml-if>
  <td><dtml-var title></td>
</tr>
</dtml-in>
</table>
```

Basic batch processing:

```
<p>
<dtml-in largeSequence size=10 start=start previous>
  <a href="<dtml-var absolute_url><dtml-var sequence-query>start=<dtml-var previous-seque
</dtml-in>

<dtml-in largeSequence size=10 start=start next>
  <a href="<dtml-var absolute_url><dtml-var sequence-query>start=<dtml-var next-sequence-
</dtml-in>
</p>

<p>
<dtml-in largeSequence size=10 start=start>
  <dtml-var sequence-item>
</dtml-in>
</p>
```

This example creates *Previous* and *Next* links to navigate between batches. Note, by using `sequence-query`, you do not lose any GET variables as you navigate between batches.

let: Defines DTML variables

The `let` tag defines variables in the DTML namespace.

Syntax

`let` tag syntax:

```
<dtml-let [Name=Variable][Name="Expression"]...>
</dtml-let>
```

The `let` tag is a block tag. Variables are defined by tag arguments. Defined variables are pushed onto the DTML namespace while the `let` block is executed. Variables are defined by attributes. The `let` tag can have one or more attributes with arbitrary names. If the attributes are defined with double quotes they are considered expressions, otherwise they are looked up by name. Attributes are processed in order, so later attributes can reference, and/or overwrite earlier ones.

Examples

Basic usage:

```
<dtml-let name="'Bob'" ids=objectIds>
  name: <dtml-var name>
  ids: <dtml-var ids>
</dtml-let>
```

Using the `let` tag with the `in` tag:

```
<dtml-in expr="(1,2,3,4)">
  <dtml-let num=sequence-item
            index=sequence-index
            result="num*index">
    <dtml-var num> * <dtml-var index> = <dtml-var result>
  </dtml-let>
</dtml-in>
```

This yields:

```
1 * 0 = 0
2 * 1 = 2
3 * 2 = 6
4 * 3 = 12
```

See Also

with tag

mime: Formats data with MIME

The `mime` tag allows you to create MIME encoded data. It is chiefly used to format email inside the `sendmail` tag.

Syntax

`mime` tag syntax:

```
<dtml-mime>
[<dtml-boundry>]
...
</dtml-mime>
```

The `mime` tag is a block tag. The block is can be divided by one or more `boundry` tags to create a multi−part MIME message. `mime` tags may be nested. The `mime` tag is most often used inside the `sendmail` tag.

Attributes

Both the `mime` and `boundry` tags have the same attributes.

encode=string
: MIME Content−Transfer−Encoding header, defaults to `base64`. Valid encoding options include `base64`, `quoted−printable`, `uuencode`, `x−uuencode`, `uue`, `x−uue`, and `7bit`. If the `encode` attribute is set to `7bit` no encoding is done on the block and the data is assumed to be in a valid MIME format.

type=string
: MIME Content−Type header.

type_expr=string

MIME Content−Type header as a variable expression. You cannot use both `type` and `type_expr`.

name=string
: MIME Content−Type header name.

name_expr=string
: MIME Content−Type header name as a variable expression. You cannot use both `name` and `name_expr`.

disposition=string
: MIME Content−Disposition header.

disposition_expr=string
: MIME Content−Disposition header as a variable expression. You cannot use both `disposition` and `disposition_expr`.

filename=string
: MIME Content−Disposition header filename.

filename_expr=string
: MIME Content−Disposition header filename as a variable expression. You cannot use both `filename` and `filename_expr`.

skip_expr=string
: A variable expression that if true, skips the block. You can use this attribute to selectively include MIME blocks.

Examples

Sending a file attachment:

```
<dtml−sendmail>
To: <dtml−recipient>
Subject: Resume
<dtml−mime type="text/plain" encode="7bit">

Hi, please take a look at my resume.

<dtml−boundary type="application/octet−stream" disposition="attachment"
encode="base64" filename_expr="resume_file.getId()"><dtml−var expr="resume_file.read()"><
</dtml−sendmail>
```

See Also

Python Library: mimetools

raise: Raises an exception

The `raise` tag raises an exception, mirroring the Python `raise` statement.

Syntax

`raise` tag syntax:

```
<dtml−raise ExceptionName|ExceptionExpression>
</dtml−raise>
```

The `raise` tag is a block tag. It raises an exception. Exceptions can be an exception class or a string. The contents of the tag are passed as the error value.

Examples

Raising a KeyError:

```
<dtml-raise KeyError></dtml-raise>
```

Raising an HTTP 404 error:

```
<dtml-raise NotFound>Web Page Not Found</dtml-raise>
```

See Also

try tag

Python Tutorial: Errors and Exceptions

Python Built−in Exceptions

return: Returns data

The `return` tag stops executing DTML and returns data. It mirrors the Python `return` statement.

Syntax

`return` tag syntax:

```
<dtml-return ReturnVariable|expr="ReturnExpression">
```

Stops execution of DTML and returns a variable or expression. The DTML output is not returned. Usually a return expression is more useful than a return variable. Scripts largely obsolete this tag.

Examples

Returning a variable:

```
<dtml-return result>
```

Returning a Python dictionary:

```
<dtml-return expr="{'hi':200, 'lo':5}">
```

sendmail: Sends email with SMTP

The `sendmail` tag sends an email message using SMTP.

Syntax

`sendmail` tag syntax:

```
<dtml-sendmail>
</dtml-sendmail>
```

The `sendmail` tag is a block tag. It either requires a `mailhost` or a `smtphost` argument, but not both. The tag block is sent as an email message. The beginning of the block describes the email headers. The headers are separated from the body by a blank line. Alternately the `To`, `From` and `Subject` headers can be set with tag arguments.

Attributes

mailhost
: The name of a Zope MailHost object to use to send email. You cannot specify both a mailhost and a smtphost.

smtphost
: The name of a SMTP server used to send email. You cannot specify both a mailhost and a smtphost.

port
: If the smtphost attribute is used, then the port attribute is used to specify a port number to connect to. If not specified, then port 25 will be used.

mailto
: The recipient address or a list of recipient addresses separated by commas. This can also be specified with the `To` header.

mailfrom
: The sender address. This can also be specified with the `From` header.

subject
: The email subject. This can also be specified with the `Subject` header.

Examples

Sending an email message using a Mail Host:

```
<dtml-sendmail mailhost="mailhost">
To: <dtml-var recipient>
From: <dtml-var sender>
Subject: <dtml-var subject>

Dear <dtml-var recipient>,

You order number <dtml-var order_number> is ready.
Please pick it up at your soonest convenience.
</dtml-sendmail>
```

See Also

RFC 821 (SMTP Protocol)

mime tag

sqlgroup: Formats complex SQL expressions

The `sqlgroup` tag formats complex boolean SQL expressions. You can use it along with the `sqltest` tag to build dynamic SQL queries that tailor themselves to the environment. This tag is used in SQL Methods.

Syntax

`sqlgroup` tag syntax:

```
<dtml-sqlgroup>
[<dtml-or>]
[<dtml-and>]
...
</dtml-sqlgroup>
```

The `sqlgroup` tag is a block tag. It is divided into blocks with one or more optional `or` and `and` tags. `sqlgroup` tags can be nested to produce complex logic.

Attributes

required=boolean
: Indicates whether the group is required. If it is not required and contains nothing, it is excluded from the DTML output.

where=boolean
: If true, includes the string "where". This is useful for the outermost `sqlgroup` tag in a SQL `select` query.

Examples

Sample usage:

```
select * from employees
<dtml-sqlgroup where>
  <dtml-sqltest salary op="gt" type="float" optional>
<dtml-and>
  <dtml-sqltest first type="nb" multiple optional>
<dtml-and>
  <dtml-sqltest last type="nb" multiple optional>
</dtml-sqlgroup>
```

If `first` is `Bob` and `last` is `Smith, McDonald` it renders:

```
select * from employees
where
(first='Bob'
 and
 last in ('Smith', 'McDonald')
)
```

If `salary` is 50000 and `last` is `Smith` it renders:

```
select * from employees
where
(salary > 50000.0
 and
 last='Smith'
)
```

Nested `sqlgroup` tags:

```
select * from employees
<dtml-sqlgroup where>
  <dtml-sqlgroup>
     <dtml-sqltest first op="like" type="nb">
  <dtml-and>
     <dtml-sqltest last op="like" type="nb">
  <dtml-sqlgroup>
```

```
<dtml-or>
  <dtml-sqltest salary op="gt" type="float">
</dtml-sqlgroup>
```

Given sample arguments, this template renders to SQL like so:

```
select * form employees
where
(
  (
   name like 'A*'
   and
   last like 'Smith'
   )
 or
 salary > 20000.0
)
```

See Also

sqltest tag

sqltest: Formats SQL condition tests

The `sqltest` tag inserts a condition test into SQL code. It tests a column against a variable. This tag is used in SQL Methods.

Syntax

`sqltest` tag syntax:

```
<dtml-sqltest Variable|expr="VariableExpression">
```

The `sqltest` tag is a singleton. It inserts a SQL condition test statement. It is used to build SQL queries. The `sqltest` tag correctly escapes the inserted variable. The named variable or variable expression is tested against a SQL column using the specified comparison operation.

Attributes

type=string
: The type of the variable. Valid types include: `string`, `int`, `float` and `nb`. `nb` means non−blank string, and should be used instead of `string` unless you want to test for blank values. The type attribute is required and is used to properly escape inserted variable.

column=string
: The name of the SQL column to test against. This attribute defaults to the variable name.

multiple=boolean
: If true, then the variable may be a sequence of values to test the column against.

optional=boolean
: If true, then the test is optional and will not be rendered if the variable is empty or non−existent.

op=string
: The comparison operation. Valid comparisons include:

  eq
  : equal to

  gt

greater than

lt
: less than

ne
: not equal to

ge
: greater than or equal to

le
: less than or equal to

The comparison defaults to equal to. If the comparison is not recognized it is used anyway. Thus you can use comparisons such as `like`.

Examples

Basic usage:

```
select * from employees
  where <dtml-sqltest name type="nb">
```

If the `name` variable is `Bob` then this renders:

```
select * from employees
  where name = 'Bob'
```

Multiple values:

```
select * from employees
  where <dtml-sqltest empid type=int multiple>
```

If the `empid` variable is `(12,14,17)` then this renders:

```
select * from employees
  where empid in (12, 14, 17)
```

See Also

sqlgroup tag

sqlvar tag

sqlvar: Inserts SQL variables

The `sqlvar` tag safely inserts variables into SQL code. This tag is used in SQL Methods.

Syntax

`sqlvar` tag syntax:

```
<dtml-sqlvar Variable|expr="VariableExpression">
```

The `sqlvar` tag is a singleton. Like the `var` tag, the `sqlvar` tag looks up a variable and inserts it. Unlike the var tag, the formatting options are tailored for SQL code.

Attributes

type=string
: The type of the variable. Valid types include: `string`, `int`, `float` and `nb`. `nb` means non−blank string and should be used in place of `string` unless you want to use blank strings. The type attribute is required and is used to properly escape inserted variable.

optional=boolean
: If true and the variable is null or non−existent, then nothing is inserted.

Examples

Basic usage:

```
select * from employees
  where name=<dtml−sqlvar name type="nb">
```

This SQL quotes the `name` string variable.

See Also

sqltest tag

tree: Inserts a tree widget

The `tree` tag displays a dynamic tree widget by querying Zope objects.

Syntax

`tree` tag syntax:

```
<dtml−tree [VariableName|expr="VariableExpression"]>
</dtml−tree>
```

The `tree` tag is a block tag. It renders a dynamic tree widget in HTML. The root of the tree is given by variable name or expression, if present, otherwise it defaults to the current object. The `tree` block is rendered for each tree node, with the current node pushed onto the DTML namespace.

Tree state is set in HTTP cookies. Thus for trees to work, cookies must be enabled. Also you can only have one tree per page.

Attributes

branches=string
: Finds tree branches by calling the named method. The default method is `tpValues` which most Zope objects support.

branches_expr=string
: Finds tree branches by evaluating the expression.

id=string
: The name of a method or id to determine tree state. It defaults to `tpId` which most Zope objects support. This attribute is for advanced usage only.

url=string

The name of a method or attribute to determine tree item URLs. It defaults to `tpURL` which most Zope objects support. This attribute is for advanced usage only.

leaves=string
: The name of a DTML Document or Method used to render nodes that don't have any children. Note: this document should begin with `<dtml-var standard_html_header>` and end with `<dtml-var standard_html_footer>` in order to ensure proper display in the tree.

header=string
: The name of a DTML Document or Method displayed before expanded nodes. If the header is not found, it is skipped.

footer=string
: The name of a DTML Document or Method displayed after expanded nodes. If the footer is not found, it is skipped.

nowrap=boolean
: If true then rather than wrap, nodes may be truncated to fit available space.

sort=string
: Sorts the branches by the named attribute.

reverse
: Reverses the order of the branches.

assume_children=boolean
: Assumes that nodes have children. This is useful if fetching and querying child nodes is a costly process. This results in plus boxes being drawn next to all nodes.

single=boolean
: Allows only one branch to be expanded at a time. When you expand a new branch, any other expanded branches close.

skip_unauthorized
: Skips nodes that the user is unauthorized to see, rather than raising an error.

urlparam=string
: A query string which is included in the expanding and contracting widget links. This attribute is for advanced usage only.

prefix=string
: Provide versions of the tag variables that start with this prefix instead of "tree", and that use underscores (_) instead of hyphens (−). The prefix must start with a letter and contain only alphanumeric characters and underscores (_).

Tag Variables

tree−item−expanded
: True if the current node is expanded.

tree−item−url
: The URL of the current node.

tree−root−url
: The URL of the root node.

tree−level
: The depth of the current node. Top−level nodes have a depth of zero.

tree−colspan
: The number of levels deep the tree is being rendered. This variable along with the `tree−level` variable can be used to calculate table rows and colspan settings when inserting table rows into the tree table.

tree−state
: The tree state expressed as a list of ids and sub−lists of ids. This variable is for advanced usage only.

Tag Control Variables

You can control the tree tag by setting these variables.

expand_all
: If this variable is true then the entire tree is expanded.

collapse_all
: If this variable is true then the entire tree is collapsed.

Examples

Display a tree rooted in the current object:

```
<dtml-tree>
  <dtml-var title_or_id>
</dtml-tree>
```

Display a tree rooted in another object, using a custom branches method:

```
<dtml-tree expr="folder.object" branches="objectValues">
  Node id : <dtml-var getId>
</dtml-tree>
```

try: Handles exceptions

The `try` tag allows exception handling in DTML, mirroring the Python `try/except` and `try/finally` constructs.

Syntax

The `try` tag has two different syntaxes, `try/except/else` and `try/finally`.

`try/except/else` Syntax:

```
<dtml-try>
<dtml-except [ExceptionName] [ExceptionName]...>
...
[<dtml-else>]
</dtml-try>
```

The `try` tag encloses a block in which exceptions can be caught and handled. There can be one or more `except` tags that handles zero or more exceptions. If an `except` tag does not specify an exception, then it handles all exceptions.

When an exception is raised, control jumps to the first `except` tag that handles the exception. If there is no `except` tag to handle the exception, then the exception is raised normally.

If no exception is raised, and there is an `else` tag, then the `else` tag will be executed after the body of the `try` tag.

The `except` and `else` tags are optional.

`try/finally` Syntax:

```
<dtml-try>
<dtml-finally>
</dtml-try>
```

The `finally` tag cannot be used in the same `try` block as the `except` and `else` tags. If there is a `finally` tag, its block will be executed whether or not an exception is raised in the `try` block.

Attributes

except
: Zero or more exception names. If no exceptions are listed then the except tag will handle all exceptions.

Tag Variables

Inside the `except` block these variables are defined.

error_type
: The exception type.

error_value
: The exception value.

error_tb
: The traceback.

Examples

Catching a math error:

```
<dtml-try>
<dtml-var expr="1/0">
<dtml-except ZeroDivisionError>
You tried to divide by zero.
</dtml-try>
```

Returning information about the handled exception:

```
<dtml-try>
<dtml-call dangerousMethod>
<dtml-except>
An error occurred.
Error type: <dtml-var error_type>
Error value: <dtml-var error_value>
</dtml-try>
```

Using finally to make sure to perform clean up regardless of whether an error is raised or not:

```
<dtml-call acquireLock>
<dtml-try>
<dtml-call someMethod>
<dtml-finally>
<dtml-call releaseLock>
</dtml-try>
```

See Also

raise tag

Python Tutorial: Errors and Exceptions

Python Built−in Exceptions

unless: Tests a condition

The `unless` tag provides a shortcut for testing negative conditions. For more complete condition testing use the `if` tag.

Syntax

`unless` tag syntax:

```
<dtml−unless ConditionVariable|expr="ConditionExpression">
</dtml−unless>
```

The `unless` tag is a block tag. If the condition variable or expression evaluates to false, then the contained block is executed. Like the `if` tag, variables that are not present are considered false.

Examples

Testing a variable:

```
<dtml−unless testMode>
  <dtml−call dangerousOperation>
</dtml−unless>
```

The block will be executed if `testMode` does not exist, or exists but is false.

See Also

if tag

var: Inserts a variable

The `var` tags allows you insert variables into DTML output.

Syntax

`var` tag syntax:

```
<dtml−var Variable|expr="Expression">
```

The `var` tag is a singleton tag. The `var` tag finds a variable by searching the DTML namespace which usually consists of current object, the current object's containers, and finally the web request. If the variable is found, it is inserted into the DTML output. If not found, Zope raises an error.

`var` tag entity syntax:

```
&dtml−variableName;
```

Entity syntax is a short cut which inserts and HTML quotes the variable. It is useful when inserting variables into HTML tags.

`var` tag entity syntax with attributes:

```
&dtml.attribute1[.attribute2]...−variableName;
```

To a limited degree you may specify attributes with the entity syntax. You may include zero or more attributes delimited by periods. You cannot provide arguments for attributes using the entity syntax. If you provide zero or more attributes, then the variable is not automatically HTML quoted. Thus you can avoid HTML quoting with this syntax, `&dtml.−variableName;`.

Attributes

html_quote
: Convert characters that have special meaning in HTML to HTML character entities.

missing=string
: Specify a default value in case Zope cannot find the variable.

fmt=string
: Format a variable. Zope provides a few built−in formats including C−style format strings. For more information on C−style format strings see the Python Library Reference If the format string is not a built−in format, then it is assumed to be a method of the object, and it called.

    whole−dollars
    : Formats the variable as dollars.

    dollars−and−cents
    : Formats the variable as dollars and cents.

    collection−length
    : The length of the variable, assuming it is a sequence.

    structured−text
    : Formats the variable as Structured Text. For more information on Structured Text see Structured Text How−To on the Zope.org web site.

null=string
: A default value to use if the variable is None.

lower
: Converts upper−case letters to lower case.

upper
: Converts lower−case letters to upper case.

capitalize
: Capitalizes the first character of the inserted word.

spacify
: Changes underscores in the inserted value to spaces.

thousands_commas
: Inserts commas every three digits to the left of a decimal point in values containing numbers for example `12000` becomes `12,000`.

url
: Inserts the URL of the object, by calling its `absolute_url` method.

url_quote
: Converts characters that have special meaning in URLs to HTML character entities.

url_quote_plus

URL quotes character, like `url_quote` but also converts spaces to plus signs.

sql_quote
: Converts single quotes to pairs of single quotes. This is needed to safely include values in SQL strings.

newline_to_br
: Convert newlines (including carriage returns) to HTML break tags.

size=arg
: Truncates the variable at the given length (Note: if a space occurs in the second half of the truncated string, then the string is further truncated to the right−most space).

etc=arg
: Specifies a string to add to the end of a string which has been truncated (by setting the `size` attribute listed above). By default, this is `...`

Examples

Inserting a simple variable into a document:

```
<dtml−var standard_html_header>
```

Truncation:

```
<dtml−var colors size=10 etc=", etc.">
```

will produce the following output if *colors* is the string 'red yellow green':

```
red yellow, etc.
```

C−style string formatting:

```
<dtml−var expr="23432.2323" fmt="%.2f">
```

renders to:

```
23432.23
```

Inserting a variable, *link*, inside an HTML `A` tag with the entity syntax:

```
<a href="&dtml−link;">Link</a>
```

Inserting a link to a document `doc`, using entity syntax with attributes:

```
<a href="&dtml.url−doc;"><dtml−var doc fmt="title_or_id"></a>
```

This creates an HTML link to an object using its URL and title. This example calls the object's `absolute_url` method for the URL (using the `url` attribute) and its `title_or_id` method for the title.

with: Controls DTML variable look up

The `with` tag pushes an object onto the DTML namespace. Variables will be looked up in the pushed object first.

Syntax

`with` tag syntax:

```
<dtml-with Variable|expr="Expression">
</dtml-with>
```

The `with` tag is a block tag. It pushes the named variable or variable expression onto the DTML namespace for the duration of the `with` block. Thus names are looked up in the pushed object first.

Attributes

only
: Limits the DTML namespace to only include the one defined in the `with` tag.

mapping
: Indicates that the variable or expression is a mapping object. This ensures that variables are looked up correctly in the mapping object.

Examples

Looking up a variable in the REQUEST:

```
<dtml-with REQUEST only>
  <dtml-if id>
    <dtml-var id>
  <dtml-else>
    'id' was not in the request.
  </dtml-if>
</dtml-with>
```

Pushing the first child on the DTML namespace:

```
<dtml-with expr="objectValues()[0]">
  First child's id: <dtml-var id>
</dtml-with>
```

See Also

let tag

Appendix B: API Reference

This reference describes the interfaces to the most common set of basic Zope objects. This reference is useful while writing DTML, Perl, and Python scripts that create and manipulate Zope objects.

module `AccessControl`

AccessControl: Security functions and classes

The functions and classes in this module are available to Python−based Scripts and Page Templates.

class `SecurityManager`

A security manager provides methods for checking access and managing executable context and policies

`calledByExecutable(self)`

Return a boolean value indicating if this context was called by an executable.

permission
: Always available

`validate(accessed=None, container=None, name=None, value=None, roles=None)`

Validate access.

Arguments:

accessed
: the object that was being accessed

container
: the object the value was found in

name
: The name used to access the value

value
: The value retrieved though the access.

roles
: The roles of the object if already known.

The arguments may be provided as keyword arguments. Some of these arguments may be omitted, however, the policy may reject access in some cases when arguments are omitted. It is best to provide all the values possible.

permission
: Always available

`checkPermission(self, permission, object)`

Check whether the security context allows the given permission on the given object.

permission
: Always available

`getUser(self)`

Get the current authenticated user. See the `AuthenticatedUser` class.

permission
: Always available

`validateValue(self, value, roles=None)`

Convenience for common case of simple value validation.

permission
: Always available

`def getSecurityManager()`

Returns the security manager. See the `SecurityManager` class.

module `AuthenticatedUser`

class `AuthenticatedUser`

This interface needs to be supported by objects that are returned by user validation and used for access control.

`getUserName()`

Return the name of a user

Permission
: Always available

`getId()`

Get the ID of the user. The ID can be used from Python to get the user from the user's UserDatabase.

Permission
: Always available

`has_role(roles, object=None)`

Return true if the user has at least one role from a list of roles, optionally in the context of an object.

Permission

Always available

`getRoles()`

Return a list of the user's roles.

Permission
: Always available

`has_permission(permission, object)`

Return true if the user has a permission on an object.

Permission
: Always available

`getRolesInContext(object)`

Return the list of roles assigned to the user, including local roles assigned in context of an object.

Permission
: Always available

`getDomains()`

Return the list of domain restrictions for a user.

Permission
: Always available

module `DTMLDocument`

class `DTMLDocument(ObjectManagerItem, PropertyManager)`

A DTML Document is a Zope object that contains and executes DTML code. It is useful to represent web pages.

`manage_edit(data, title)`

Change the DTML Document, replacing its contents with `data` and changing its title.

The data argument may be a file object or a string.

Permission
: `Change DTML Documents`

`document_src()`

Returns the unrendered source text of the DTML Document.

Permission
: `View management screens`

`__call__(client=None, REQUEST={}, RESPONSE=None, **kw)`

Calling a DTMLDocument causes the Document to interpret the DTML code that it contains. The method returns the result of the interpretation, which can be any kind of object.

To accomplish its task, DTML Document often needs to resolve various names into objects. For example, when the code '<dtml−var spam>' is executed, the DTML engine tries to resolve the name `spam`.

In order to resolve names, the Document must be passed a namespace to look them up in. This can be done several ways:

- By passing a `client` object −− If the argument `client` is passed, then names are looked up as attributes on the argument.
- By passing a `REQUEST` mapping −− If the argument `REQUEST` is passed, then names are looked up as items on the argument. If the object is not a mapping, an TypeError will be raised when a name lookup is attempted.
- By passing keyword arguments −− names and their values can be passed as keyword arguments to the Document.

The namespace given to a DTML Document is the composite of these three methods. You can pass any number of them or none at all. Names are looked up first in the keyword arguments, then in the client, and finally in the mapping.

A DTMLDocument implicitly pass itself as a client argument in addition to the specified client, so names are looked up in the DTMLDocument itself.

Passing in a namespace to a DTML Document is often referred to as providing the Document with a *context*.

DTML Documents can be called three ways.

From DTML

A DTML Document can be called from another DTML Method or Document:

```
<dtml−var standard_html_header>
  <dtml−var aDTMLDocument>
<dtml−var standard_html_footer>
```

In this example, the Document `aDTMLDocument` is being called from another DTML object by name. The calling method passes the value `this` as the client argument and the current DTML namespace as the REQUEST argument. The above is identical to this following usage in a DTML Python expression:

```
<dtml−var standard_html_header>
  <dtml−var "aDTMLDocument(_.None, _)">
<dtml−var standard_html_footer>
```

From Python

Products, External Methods, and Scripts can call a DTML Document in the same way as calling a DTML Document from a Python expression in DTML; as shown in the previous example.

By the Publisher

When the URL of a DTML Document is fetched from Zope, the DTML Document is called by the publisher. The REQUEST object is passed as the second argument to the Document.

Permission
    `View`

`get_size()`

Returns the size of the unrendered source text of the DTML Document in bytes.

Permission
    `View`

ObjectManager Constructor

`manage_addDocument(id, title)`

Add a DTML Document to the current ObjectManager

module `DTMLMethod`

class `DTMLMethod(ObjectManagerItem)`

A DTML Method is a Zope object that contains and executes DTML code. It can act as a template to display other objects. It can also hold small pieces of content which are inserted into other DTML Documents or DTML Methods.

The DTML Method's id is available via the `document_id` variable and the title is available via the `document_title` variable.

`manage_edit(data, title)`

Change the DTML Method, replacing its contents with `data` and changing its title.

The data argument may be a file object or a string.

Permission
    `Change DTML Methods`

`document_src()`

Returns the unrendered source text of the DTML Method.

Permission
    `View management screens`

`__call__(client=None, REQUEST={}, **kw)`

Calling a DTMLMethod causes the Method to interpret the DTML code that it contains. The method returns the result of the interpretation, which can be any kind of object.

To accomplish its task, DTML Method often needs to resolve various names into objects. For example, when the code '<dtml−var spam>' is executed, the DTML engine tries to resolve the name `spam`.

In order to resolve names, the Method must be passed a namespace to look them up in. This can be done several ways:

- By passing a `client` object −− If the argument `client` is passed, then names are looked up as attributes on the argument.
- By passing a `REQUEST` mapping −− If the argument `REQUEST` is passed, then names are looked up as items on the argument. If the object is not a mapping, an TypeError will be raised when a name lookup is attempted.
- By passing keyword arguments −− names and their values can be passed as keyword arguments to the Method.

The namespace given to a DTML Method is the composite of these three methods. You can pass any number of them or none at all. Names will be looked up first in the keyword argument, next in the client and finally in the mapping.

Unlike DTMLDocuments, DTMLMethods do not look up names in their own instance dictionary.

Passing in a namespace to a DTML Method is often referred to as providing the Method with a *context*.

DTML Methods can be called three ways:

From DTML

A DTML Method can be called from another DTML Method or Document:

```
<dtml−var standard_html_header>
  <dtml−var aDTMLMethod>
<dtml−var standard_html_footer>
```

In this example, the Method `aDTMLMethod` is being called from another DTML object by name. The calling method passes the value `this` as the client argument and the current DTML namespace as the REQUEST argument. The above is identical to this following usage in a DTML Python expression:

```
<dtml−var standard_html_header>
  <dtml−var "aDTMLMethod(_.None, _)">
<dtml−var standard_html_footer>
```

From Python

Products, External Methods, and Scripts can call a DTML Method in the same way as calling a DTML Method from a Python expression in DTML; as shown in the previous example.

By the Publisher

When the URL of a DTML Method is fetched from Zope, the DTML Method is called by the publisher. The REQUEST object is passed as the second argument to the Method.

Permission
    `View`

`get_size()`

Returns the size of the unrendered source text of the DTML Method in bytes.

Permission
    `View`

ObjectManager Constructor

`manage_addDTMLMethod(id, title)`

Add a DTML Method to the current ObjectManager

module `DateTime`

class `DateTime`

The DateTime object provides an interface for working with dates and times in various formats. DateTime also provides methods for calendar operations, date and time arithmetic and formatting.

DateTime objects represent instants in time and provide interfaces for controlling its representation without affecting the absolute value of the object.

DateTime objects may be created from a wide variety of string or numeric data, or may be computed from other DateTime objects. DateTimes support the ability to convert their representations to many major timezones, as well as the ability to create a DateTime object in the context of a given timezone.

DateTime objects provide partial numerical behavior:

- Two date−time objects can be subtracted to obtain a time, in days between the two.
- A date−time object and a positive or negative number may be added to obtain a new date−time object that is the given number of days later than the input date−time object.
- A positive or negative number and a date−time object may be added to obtain a new date−time object that is the given number of days later than the input date−time object.
- A positive or negative number may be subtracted from a date−time object to obtain a new date−time object that is the given number of days earlier than the input date−time object.

DateTime objects may be converted to integer, long, or float numbers of days since January 1, 1901, using the standard int, long, and float functions (Compatibility Note: int, long and float return the number of days since 1901 in GMT rather than local machine timezone). DateTime objects also provide access to their value in a float format usable with the python time module, provided that the value of the object falls in the range of the epoch−based time module.

A DateTime object should be considered immutable; all conversion and numeric operations return a new DateTime object rather than modify the current object.

A DateTime object always maintains its value as an absolute UTC time, and is represented in the context of some timezone based on the arguments used to create the object. A DateTime object's methods return values based on the timezone context.

Note that in all cases the local machine timezone is used for representation if no timezone is specified.

DateTimes may be created with from zero to seven arguments.

- If the function is called with no arguments, then the current date/time is returned, represented in the timezone of the local machine.
- If the function is invoked with a single string argument which is a recognized timezone name, an object representing the current time is returned, represented in the specified timezone.
- If the function is invoked with a single string argument representing a valid date/time, an object representing that date/time will be returned.

  As a general rule, any date−time representation that is recognized and unambiguous to a resident of North America is acceptable.(The reason for this qualification is that in North America, a date like: 2/1/1994 is interpreted as February 1, 1994, while in some parts of the world, it is interpreted as January 2, 1994.) A date/time string consists of two components, a date component and an optional time component, separated by one or more spaces. If the time component is omitted, 12:00am is assumed. Any recognized timezone name specified as the final element of the date/time string will be used for computing the date/time value. (If you create a DateTime with the string `Mar 9, 1997 1:45pm US/Pacific`, the value will essentially be the same as if you had captured time.time() at the specified date and time on a machine in that timezone):

  ```
  e=DateTime("US/Eastern")
  # returns current date/time, represented in US/Eastern.

  x=DateTime("1997/3/9 1:45pm")
  # returns specified time, represented in local machine zone.

  y=DateTime("Mar 9, 1997 13:45:00")
  # y is equal to x
  ```

  The date component consists of year, month, and day values. The year value must be a one−, two−, or four−digit integer. If a one− or two−digit year is used, the year is assumed to be in the twentieth century. The month may be an integer, from 1 to 12, a month name, or a month abbreviation, where a period may optionally follow the abbreviation. The day must be an integer from 1 to the number of days in the month. The year, month, and day values may be separated by periods, hyphens, forward slashes, or spaces. Extra spaces are permitted around the delimiters. Year, month, and day values may be given in any order as long as it is possible to distinguish the components. If all three components are numbers that are less than 13, then a month−day−year ordering is assumed.

  The time component consists of hour, minute, and second values separated by colons. The hour value must be an integer between 0 and 23 inclusively. The minute value must be an integer between 0 and 59 inclusively. The second value may be an integer value between 0 and 59.999 inclusively. The second value or both the minute and second values may be omitted. The time may be followed by am or pm in upper or lower case, in which case a 12−hour clock is assumed.

- If the DateTime function is invoked with a single Numeric argument, the number is assumed to be a floating point value such as that returned by time.time().

  A DateTime object is returned that represents the gmt value of the time.time() float represented in the local machine's timezone.

- If the function is invoked with two numeric arguments, then the first is taken to be an integer year and the second argument is taken to be an offset in days from the beginning of the year, in the context of the local machine timezone. The date−time value returned is the given offset number of days from the beginning of the given year, represented in the timezone of the local machine. The offset may be positive or negative. Two−digit years are assumed to be in the twentieth century.

- If the function is invoked with two arguments, the first a float representing a number of seconds past the epoch in gmt (such as those returned by time.time()) and the second a string naming a recognized timezone, a DateTime with a value of that gmt time will be returned, represented in the given timezone.:

  ```
  import time
  t=time.time()

  now_east=DateTime(t,'US/Eastern')
  # Time t represented as US/Eastern

  now_west=DateTime(t,'US/Pacific')
  # Time t represented as US/Pacific

  # now_east == now_west
  # only their representations are different
  ```

- If the function is invoked with three or more numeric arguments, then the first is taken to be an integer year, the second is taken to be an integer month, and the third is taken to be an integer day. If the combination of values is not valid, then a DateTimeError is raised. Two−digit years are assumed to be in the twentieth century. The fourth, fifth, and sixth arguments specify a time in hours, minutes, and seconds; hours and minutes should be positive integers and seconds is a positive floating point value, all of these default to zero if not given. An optional string may be given as the final argument to indicate timezone (the effect of this is as if you had taken the value of time.time() at that time on a machine in the specified timezone).

If a string argument passed to the DateTime constructor cannot be parsed, it will raise DateTime.SyntaxError. Invalid date, time, or timezone components will raise a DateTime.DateTimeError.

The module function Timezones() will return a list of the timezones recognized by the DateTime module. Recognition of timezone names is case−insensitive.

`strftime(format)`

Return date time string formatted according to `format`

See Python's time.strftime function.

`dow()`

Return the integer day of the week, where Sunday is 0

Permission
: Always available

`aCommon()`

Return a string representing the object's value in the format: Mar 1, 1997 1:45 pm

Permission
: Always available

`h_12()`

Return the 12−hour clock representation of the hour

Permission
: Always available

`Mon_()`

Compatibility: see pMonth

Permission
: Always available

`HTML4()`

Return the object in the format used in the HTML4.0 specification, one of the standard forms in ISO8601.

See HTML 4.0 Specification

Dates are output as: YYYY−MM−DDTHH:MM:SSZ T, Z are literal characters. The time is in UTC.

Permission
: Always available

`greaterThanEqualTo(t)`

Compare this DateTime object to another DateTime object OR a floating point number such as that which is returned by the python time module. Returns true if the object represents a date/time greater than or equal to the specified DateTime or time module style time. Revised to give more correct results through comparison of long integer milliseconds.

Permission
: Always available

`dayOfYear()`

Return the day of the year, in context of the timezone representation of the object

Permission
: Always available

`lessThan(t)`

Compare this DateTime object to another DateTime object OR a floating point number such as that which is returned by the python time module. Returns true if the object represents a date/time less than the specified DateTime or time module style time. Revised to give more correct results through comparison of long integer milliseconds.

Permission
: Always available

`AMPM()`

Return the time string for an object to the nearest second.

Permission
: Always available

`isCurrentHour()`

Return true if this object represents a date/time that falls within the current hour, in the context of this object's timezone representation

Permission
: Always available

`Month()`

Return the full month name

Permission
: Always available

`mm()`

Return month as a 2 digit string

Permission
: Always available

`ampm()`

Return the appropriate time modifier (am or pm)

Permission
: Always available

`hour()`

Return the 24−hour clock representation of the hour

Permission
: Always available

`aCommonZ()`

Return a string representing the object's value in the format: Mar 1, 1997 1:45 pm US/Eastern

Permission
: Always available

`Day_()`

Compatibility: see pDay

Permission
: Always available

`pCommon()`

Return a string representing the object's value in the format: Mar. 1, 1997 1:45 pm

Permission
: Always available

`minute()`

Return the minute

Permission
: Always available

`day()`

Return the integer day

Permission
: Always available

`earliestTime()`

Return a new DateTime object that represents the earliest possible time (in whole seconds) that still falls within the current object's day, in the object's timezone context

Permission
: Always available

`Date()`

Return the date string for the object.

Permission
: Always available

`Time()`

Return the time string for an object to the nearest second.

Permission
: Always available

`isFuture()`

Return true if this object represents a date/time later than the time of the call

Permission
: Always available

`greaterThan(t)`

Compare this DateTime object to another DateTime object OR a floating point number such as that which is returned by the python time module. Returns true if the object represents a date/time greater than the specified DateTime or time module style time. Revised to give more correct results through comparison of long integer milliseconds.

Permission
: Always available

`TimeMinutes()`

Return the time string for an object not showing seconds.

Permission
: Always available

`yy()`

Return calendar year as a 2 digit string

Permission
: Always available

`isCurrentDay()`

Return true if this object represents a date/time that falls within the current day, in the context of this object's timezone representation

Permission
: Always available

`dd()`

Return day as a 2 digit string

Permission
: Always available

`rfc822()`

Return the date in RFC 822 format

Permission
: Always available

`isLeapYear()`

Return true if the current year (in the context of the object's timezone) is a leap year

Permission
: Always available

`fCommon()`

Return a string representing the object's value in the format: March 1, 1997 1:45 pm

Permission
: Always available

`isPast()`

Return true if this object represents a date/time earlier than the time of the call

Permission
: Always available

`fCommonZ()`

Return a string representing the object's value in the format: March 1, 1997 1:45 pm US/Eastern

Permission
: Always available

`timeTime()`

Return the date/time as a floating−point number in UTC, in the format used by the python time module. Note that it is possible to create date/time values with DateTime that have no meaningful value to the time module.

Permission
: Always available

`toZone(z)`

Return a DateTime with the value as the current object, represented in the indicated timezone.

Permission
: Always available

`lessThanEqualTo(t)`

Compare this DateTime object to another DateTime object OR a floating point number such as that which is returned by the python time module. Returns true if the object represents a date/time less than or equal to the specified DateTime or time module style time. Revised to give more correct results through comparison of long integer milliseconds.

Permission
: Always available

`Mon()`

Compatibility: see aMonth

Permission
: Always available

`parts()`

Return a tuple containing the calendar year, month, day, hour, minute second and timezone of the object

Permission
: Always available

`isCurrentYear()`

Return true if this object represents a date/time that falls within the current year, in the context of this object's timezone representation

Permission
: Always available

`PreciseAMPM()`

Return the time string for the object.

Permission
: Always available

`AMPMMinutes()`

Return the time string for an object not showing seconds.

Permission
: Always available

`equalTo(t)`

Compare this DateTime object to another DateTime object OR a floating point number such as that which is returned by the python time module. Returns true if the object represents a date/time equal to the specified DateTime or time module style time. Revised to give more correct results through comparison of long integer milliseconds.

Permission
: Always available

`pDay()`

Return the abbreviated (with period) name of the day of the week

Permission
: Always available

`notEqualTo(t)`

Compare this DateTime object to another DateTime object OR a floating point number such as that which is returned by the python time module. Returns true if the object represents a date/time not equal to the specified DateTime or time module style time. Revised to give more correct results through comparison of long integer milliseconds.

Permission
Always available

`h_24()`

Return the 24−hour clock representation of the hour

Permission
Always available

`pCommonZ()`

Return a string representing the object's value in the format: Mar. 1, 1997 1:45 pm US/Eastern

Permission
Always available

`isCurrentMonth()`

Return true if this object represents a date/time that falls within the current month, in the context of this object's timezone representation

Permission
Always available

`DayOfWeek()`

Compatibility: see Day

Permission
Always available

`latestTime()`

Return a new DateTime object that represents the latest possible time (in whole seconds) that still falls within the current object's day, in the object's timezone context

Permission
Always available

`dow_1()`

Return the integer day of the week, where Sunday is 1

Permission

Always available

`timezone()`

Return the timezone in which the object is represented.

Permission
: Always available

`year()`

Return the calendar year of the object

Permission
: Always available

`PreciseTime()`

Return the time string for the object.

Permission
: Always available

`ISO()`

Return the object in ISO standard format

Dates are output as: YYYY−MM−DD HH:MM:SS

Permission
: Always available

`millis()`

Return the millisecond since the epoch in GMT.

Permission
: Always available

`second()`

Return the second

Permission
: Always available

`month()`

Return the month of the object as an integer

Permission
: Always available

`pMonth()`

Return the abbreviated (with period) month name.

Permission
: Always available

`aMonth()`

Return the abbreviated month name.

Permission
: Always available

`isCurrentMinute()`

Return true if this object represents a date/time that falls within the current minute, in the context of this object's timezone representation

Permission
: Always available

`Day()`

Return the full name of the day of the week

Permission
: Always available

`aDay()`

Return the abbreviated name of the day of the week

Permission
: Always available

module `ExternalMethod`

class `ExternalMethod`

Web−callable functions that encapsulate external Python functions.

The function is defined in an external file. This file is treated like a module, but is not a module. It is not imported directly, but is rather read and evaluated. The file must reside in the `Extensions` subdirectory of the Zope installation, or in an `Extensions` subdirectory of a product directory.

Due to the way ExternalMethods are loaded, it is not *currently* possible to import Python modules that reside in the `Extensions` directory. It is possible to import modules found in the `lib/python` directory of the Zope installation, or in packages that are in the `lib/python` directory.

`manage_edit(title, module, function, REQUEST=None)`

Change the External Method.

See the description of manage_addExternalMethod for a description of the arguments `module` and `function`.

Note that calling `manage_edit` causes the "module" to be effectively reloaded. This is useful during debugging to see the effects of changes, but can lead to problems of functions rely on shared global data.

`__call__(*args, **kw)`

Call the External Method.

Calling an External Method is roughly equivalent to calling the original actual function from Python. Positional and keyword parameters can be passed as usual. Note however that unlike the case of a normal Python method, the "self" argument must be passed explicitly. An exception to this rule is made if:

- The supplied number of arguments is one less than the required number of arguments, and
- The name of the function's first argument is `self`.

In this case, the URL parent of the object is supplied as the first argument.

ObjectManager Constructor

`manage_addExternalMethod(id, title, module, function)`

Add an external method to an `ObjectManager`.

In addition to the standard object−creation arguments, `id` and title, the following arguments are defined:

function
: The name of the python function. This can be a an ordinary Python function, or a bound method.

module
: The name of the file containing the function definition.

The module normally resides in the `Extensions` directory, however, the file name may have a prefix of `product.`, indicating that it should be found in a product directory.

For example, if the module is: `ACMEWidgets.foo`, then an attempt will first be made to use the file `lib/python/Products/ACMEWidgets/Extensions/foo.py`. If this failes, then the file `Extensions/ACMEWidgets.foo.py` will be used.

module `File`

class `File(ObjectManagerItem, PropertyManager)`

A File is a Zope object that contains file content. A File object can be used to upload or download file information with Zope.

Using a File object in Zope is easy. The most common usage is to display the contents of a file object in a web page. This is done by simply referencing the object from DTML:

```
        <dtml-var standard_html_header>
          <dtml-var FileObject>
        <dtml-var standard_html_footer>
```

A more complex example is presenting the File object for download by the user. The next example displays a link to every File object in a folder for the user to download:

```
        <dtml-var standard_html_header>
        <ul>
          <dtml-in "ObjectValues('File')">
            <li><a href="<dtml-var absolute_url>"><dtml-var
            id></a></li>
          </dtml-in>
        </ul>
        <dtml-var standard_html_footer>
```

In this example, the `absolute_url` method and `id` are used to create a list of HTML hyperlinks to all of the File objects in the current Object Manager.

Also see ObjectManager for details on the `objectValues` method.

getContentType()

Returns the content type of the file.

Permission
`View`

update_data(data, content_type=None, size=None)

Updates the contents of the File with `data`.

The `data` argument must be a string. If `content_type` is not provided, then a content type will not be set. If size is not provided, the size of the file will be computed from `data`.

Permission
Python only

getSize()

Returns the size of the file in bytes.

Permission
`View`

ObjectManager Constructor

manage_addFile(id, file="", title="", precondition="", content_type="")

Add a new File object.

Creates a new File object `id` with the contents of `file`

module `Folder`

class `Folder(ObjectManagerItem, ObjectManager, PropertyManager)`

A Folder is a generic container object in Zope.

Folders are the most common ObjectManager subclass in Zope.

ObjectManager Constructor

`manage_addFolder(id, title)`

Add a Folder to the current ObjectManager

Permission
    `Add Folders`

module `Image`

class `Image(File)`

An Image is a Zope object that contains image content. An Image object can be used to upload or download image information with Zope.

Image objects have two properties the define their dimension, `height` and `width`. These are calculated when the image is uploaded. For image types that Zope does not understand, these properties may be undefined.

Using a Image object in Zope is easy. The most common usage is to display the contents of an image object in a web page. This is done by simply referencing the object from DTML:

```
<dtml-var standard_html_header>
  <dtml-var ImageObject>
<dtml-var standard_html_footer>
```

This will generate an HTML IMG tag referencing the URL to the Image. This is equivalent to:

```
<dtml-var standard_html_header>
  <dtml-with ImageObject>
    <img src="<dtml-var absolute_url>">
  </dtml-with>
<dtml-var standard_html_footer>
```

You can control the image display more precisely with the `tag` method. For example:

```
<dtml-var "ImageObject.tag(border='5', align='left')">
```

`tag(height=None, width=None, alt=None, scale=0, xscale=0, yscale=0, **args)`

This method returns a string which contains an HTML IMG tag reference to the image.

Optionally, the `height`, `width`, `alt`, `scale`, `xscale` and `yscale` arguments can be provided which are turned into HTML IMG tag attributes. Note, `height` and `width` are provided by default, and `alt` comes from the `title_or_id` method.

Keyword arguments may be provided to support other or future IMG tag attributes. The one exception to this is the HTML Cascading Style Sheet tag `class`. Because the word `class` is a reserved keyword in Python, you must instead use the keyword argument `css_class`. This will be turned into a `class` HTML tag attribute on the rendered `img` tag.

Permission
: `View`

ObjectManager Constructor

`manage_addImage(id, file, title="", precondition="", content_type="")`

Add a new Image object.

Creates a new Image object `id` with the contents of `file`.

module `MailHost`

class `MailHost`

MailHost objects work as adapters to Simple Mail Transfer Protocol (SMTP) servers. MailHosts are used by DTML `sendmail` tags to find the proper host to deliver mail to.

`send(messageText, mto=None, mfrom=None, subject=None, encode=None)`

Sends an email message. The arguments are:

messageText
: The body of the mail message.

mto
: A string or list of recipient(s) of the message.

mfrom
: The address of the message sender.

subject
: The subject of the message.

encode
: The rfc822 defined encoding of the message. The default of `None` means no encoding is done. Valid values are `base64`, `quoted-printable` and `uuencode`.

ObjectManager Constructor

`manage_addMailHost(id, title="", smtp_host=None, localhost=localhost, smtp_port=25, timeout=1.0)`

Add a mailhost object to an ObjectManager.

module `ObjectManager`

class `ObjectManager`

An ObjectManager contains other Zope objects. The contained objects are Object Manager Items.

To create an object inside an object manager use `manage_addProduct`:

```
        self.manage_addProduct['OFSP'].manage_addFolder(id, title)
```

In DTML this would be:

```
          <dtml-call "manage_addProduct['OFSP'].manage_addFolder(id, title)">
```

These examples create a new Folder inside the current ObjectManager.

`manage_addProduct` is a mapping that provides access to product constructor methods. It is indexed by product id.

Constructor methods are registered during product initialization and should be documented in the API docs for each addable object.

`objectItems(type=None)`

This method returns a sequence of (id, object) tuples.

Like objectValues and objectIds, it accepts one argument, either a string or a list to restrict the results to objects of a given meta_type or set of meta_types.

Each tuple's first element is the id of an object contained in the Object Manager, and the second element is the object itself.

Example:

```
          <dtml-in objectItems>
           id: <dtml-var sequence-key>,
           type: <dtml-var meta_type>
          <dtml-else>
            There are no sub-objects.
          </dtml-in>
```

Permission
      `Access contents information`

`superValues(type)`

This method returns a list of objects of a given meta_type(es) contained in the Object Manager and all its parent Object Managers.

The type argument specifies the meta_type(es). It can be a string specifying one meta_type, or it can be a list of strings to specify many.

Permission

Python only

`objectValues(type=None)`

This method returns a sequence of contained objects.

Like objectItems and objectIds, it accepts one argument, either a string or a list to restrict the results to objects of a given meta_type or set of meta_types.

Example:

```
<dtml-in expr="objectValues('Folder')">
  <dtml-var icon>
  This is the icon for the: <dtml-var id> Folder<br>.
<dtml-else>
  There are no Folders.
</dtml-in>
```

The results were restricted to Folders by passing a meta_type to `objectValues` method.

Permission
: `Access contents information`

`objectIds(type=None)`

This method returns a list of the ids of the contained objects.

Optionally, you can pass an argument specifying what object meta_type(es) to restrict the results to. This argument can be a string specifying one meta_type, or it can be a list of strings to specify many.

Example:

```
<dtml-in objectIds>
  <dtml-var sequence-item>
<dtml-else>
  There are no sub-objects.
</dtml-in>
```

This DTML code will display all the ids of the objects contained in the current Object Manager.

Permission
: `Access contents information`

module `ObjectManagerItem`

class `ObjectManagerItem`

A Zope object that can be contained within an Object Manager. Almost all Zope objects that can be managed through the web are Object Manager Items.

ObjectMangerItems have these instance attributes:

`title`
: The title of the object.

This is an optional one−line string description of the object.

`meta_type`
: A short name for the type of the object.
This is the name that shows up in product add list for the object and is used when filtering objects by type.

  This attribute is provided by the object's class and should not be changed directly.

`REQUEST`
: The current web request.
This object is acquired and should not be set.

`title_or_id()`

If the title is not blank, return it, otherwise return the id.

Permission
: Always available

`getPhysicalRoot()`

Returns the top−level Zope Application object.

Permission
: Python only

`manage_workspace()`

This is the web method that is called when a user selects an item in a object manager contents view or in the Zope Management navigation view.

Permission
: `View management screens`

`getPhysicalPath()`

Get the path of an object from the root, ignoring virtual hosts.

Permission
: Always available

`unrestrictedTraverse(path, default=None)`

Return the object obtained by traversing the given path from the object on which the method was called. This method begins with "unrestricted" because (almost) no security checks are performed.

If an object is not found then the `default` argument will be returned.

Permission
: Python only

`getId()`

Returns the object's id.

The `id` is the unique name of the object within its parent object manager. This should be a string, and can contain letters, digits, underscores, dashes, commas, and spaces.

This method replaces direct access to the `id` attribute.

Permission
: Always available

`absolute_url(relative=None)`

Return the absolute url to the object.

If the relative argument is provided with a true value, then the URL returned is relative to the site object. Note, if virtual hosts are being used, then the path returned is a logical, rather than a physical path.

Permission
: Always available

`this()`

Return the object.

This turns out to be handy in two situations. First, it provides a way to refer to an object in DTML expressions.

The second use for this is rather deep. It provides a way to acquire an object without getting the full context that it was acquired from. This is useful, for example, in cases where you are in a method of a non−item subobject of an item and you need to get the item outside of the context of the subobject.

Permission
: Always available

`restrictedTraverse(path, default=None)`

Return the object obtained by traversing the given path from the object on which the method was called, performing security checks along the way.

If an object is not found then the `default` argument will be returned.

Permission
: Always available

`title_and_id()`

If the title is not blank, the return the title followed by the id in parentheses. Otherwise return the id.

Permission
: Always available

module `PropertyManager`

class `PropertyManager`

A Property Manager object has a collection of typed attributes called properties. Properties can be managed through the web or via DTML.

In addition to having a type, properties can be writable or read−only and can have default values.

`propertyItems()`

Return a list of (id, property) tuples.

Permission
    `Access contents information`

`propertyValues()`

Returns a list of property values.

Permission
    `Access contents information`

`propertyMap()`

Returns a tuple of mappings, giving meta−data for properties. The meta−data includes `id`, `type`, and `mode`.

Permission
    `Access contents information`

`propertyIds()`

Returns a list of property ids.

Permission
    `Access contents information`

`getPropertyType(id)`

Get the type of property `id`. Returns None if no such property exists.

Permission
    `Access contents information`

`getProperty(id, d=None)`

Return the value of the property `id`. If the property is not found the optional second argument or None is returned.

Permission
    `Access contents information`

`hasProperty(id)`

Returns a true value if the Property Manager has the property `id`. Otherwise returns a false value.

Permission
    `Access contents information`

module `PropertySheet`

class `PropertySheet`

A PropertySheet is an abstraction for organizing and working with a set of related properties. Conceptually it acts like a container for a set of related properties and meta−data describing those properties. A PropertySheet may or may not provide a web interface for managing its properties.

`xml_namespace()`

Return a namespace string usable as an xml namespace for this property set. This may be an empty string if there is no default namespace for a given property sheet (especially property sheets added in ZClass definitions).

Permission
    Python only

`propertyItems()`

Return a list of (id, property) tuples.

Permission
    `Access contents information`

`propertyValues()`

Returns a list of actual property values.

Permission
    `Access contents information`

`getPropertyType(id)`

Get the type of property `id`. Returns None if no such property exists.

Permission
    Python only

`propertyInfo()`

Returns a mapping containing property meta−data.

Permission
    Python only

`getProperty(id, d=None)`

Get the property `id`, returning the optional second argument or None if no such property is found.

Permission
: Python only

`manage_delProperties(ids=None, REQUEST=None)`

Delete one or more properties with the given `ids`. The `ids` argument should be a sequence (tuple or list) containing the ids of the properties to be deleted. If `ids` is empty no action will be taken. If any of the properties named in `ids` does not exist, an error will be raised.

Some objects have "special" properties defined by product authors that cannot be deleted. If one of these properties is named in `ids`, an HTML error message is returned.

If no value is passed in for `REQUEST`, the method will return None. If a value is provided for `REQUEST` (as it will be when called via the web), the property management form for the object will be rendered and returned.

This method may be called via the web, from DTML or from Python code.

Permission
: `Manage Properties`

`manage_changeProperties(REQUEST=None, **kw)`

Change existing object properties by passing either a mapping object as `REQUEST` containing name:value pairs or by passing name=value keyword arguments.

Some objects have "special" properties defined by product authors that cannot be changed. If you try to change one of these properties through this method, an error will be raised.

Note that no type checking or conversion happens when this method is called, so it is the caller's responsibility to ensure that the updated values are of the correct type. *This should probably change*.

If a value is provided for `REQUEST` (as it will be when called via the web), the method will return an HTML message dialog. If no REQUEST is passed, the method returns `None` on success.

This method may be called via the web, from DTML or from Python code.

Permission
: `Manage Properties`

`manage_addProperty(id, value, type, REQUEST=None)`

Add a new property with the given `id`, `value` and `type`.

These are the property types:

`boolean`
: 1 or 0.

`date`

A `DateTime` value, for example `12/31/1999 15:42:52 PST`.

`float`
: A decimal number, for example `12.4`.

`int`
: An integer number, for example, `12`.

`lines`
: A list of strings, one per line.

`long`
: A long integer, for example `12232322322323232323423`.

`string`
: A string of characters, for example `This is a string`.

`text`
: A multi−line string, for example a paragraph.

`tokens`
: A list of strings separated by white space, for example `one two three`.

`selection`
: A string selected by a pop−up menu.

`multiple selection`
: A list of strings selected by a selection list.

This method will use the passed in `type` to try to convert the `value` argument to the named type. If the given `value` cannot be converted, a ValueError will be raised.

The value given for `selection` and `multiple selection` properites may be an attribute or method name. The attribute or method must return a sequence values.

If the given `type` is not recognized, the `value` and `type` given are simply stored blindly by the object.

If no value is passed in for `REQUEST`, the method will return `None`. If a value is provided for `REQUEST` (as it will when called via the web), the property management form for the object will be rendered and returned.

This method may be called via the web, from DTML or from Python code.

Permission
: `Manage Properties`

`propertyMap()`

Returns a tuple of mappings, giving meta−data for properties.

Permssion
: Python only

`propertyIds()`

Returns a list of property ids.

Permission
: `Access contents information`

`hasProperty(id)`

Returns true if `self` has a property with the given `id`, false otherwise.

Permission
: `Access contents information`

module `PropertySheets`

class `PropertySheets`

A PropertySheet is an abstraction for organizing and working with a set of related properties. Conceptually it acts like a container for a set of related properties and meta−data describing those properties. PropertySheet objects are accessed through a PropertySheets object that acts as a collection of PropertySheet instances.

Objects that support property sheets (objects that support the PropertyManager interface or ZClass objects) have a `propertysheets` attribute (a PropertySheets instance) that is the collection of PropertySheet objects. The PropertySheets object exposes an interface much like a Python mapping, so that individual PropertySheet objects may be accessed via dictionary−style key indexing.

`get(name, default=None)`

Return the PropertySheet identified by `name`, or the value given in `default` if the named PropertySheet is not found.

Permission
: Python only

`values()`

Return a sequence of all of the PropertySheet objects in the collection.

Permission
: Python only

`items()`

Return a sequence containing an `(id, object)` tuple for each PropertySheet object in the collection.

Permission
: Python only

module `PythonScript`

class `PythonScript(Script)`

Python Scripts contain python code that gets executed when you call the script by:

- Calling the script through the web by going to its location with a web browser.
- Calling the script from another script object.
- Calling the script from a method object, such as a DTML Method.

Python Scripts can contain a "safe" subset of the python language. Python Scripts must be safe because they can be potentially edited by many different users through an insecure medium like the web. The following safety issues drive the need for secure Python Scripts:

- Because many users can use Zope, a Python Script must make sure it does not allow a user to do something they are not allowed to do, like deleting an object they do not have permission to delete. Because of this requirement, Python Scripts do many security checks in the course of their execution.
- Because Python Scripts can be edited through the insecure medium of the web, they are not allowed access to the Zope server's file−system. Normal Python builtins like `open` are, therefore, not allowed.
- Because many standard Python modules break the above two security restrictions, only a small subset of Python modules may be imported into a Python Scripts with the "import" statement unless they have been validated by Zope's security policy. Currently, the following standard python modules have been validated:
    - string
    - math
    - whrandom and random
    - Products.PythonScripts.standard
- Because it allows you to execute arbitrary python code, the python "exec" statement is not allowed in Python methods.
- Because they may represent or cause security violations, some Python builtin functions are not allowed. The following Python builtins are not allowed:
    - open
    - input
    - raw_input
    - eval
    - execfile
    - compile
    - type
    - coerce
    - intern
    - dir
    - globals
    - locals
    - vars
    - buffer
    - reduce
- Other builtins are restricted in nature. The following builtins are restricted:

  range
  : Due to possible memory denial of service attacks, the range builtin is restricted to creating ranges less than 10,000 elements long.

  filter, map, tuple, list
  : For the same reason, builtins that construct lists from sequences do not operate on strings.

  getattr, setattr, delattr
  : Because these may enable Python code to circumvent Zope's security system, they are replaced with custom, security constrained versions.
- In order to be consistent with the Python expressions available to DTML, the builtin functions are augmented with a small number of functions and a class:
    - test
    - namespace
    - render
    - same_type
    - DateTime

- Because the "print" statement cannot operate normally in Zope, its effect has been changed. Rather than sending text to stdout, "print" appends to an internal variable. The special builtin name "printed" evaluates to the concatenation of all text printed so far during the current execution of the script.

`document_src(REQUEST=None, RESPONSE=None)`

Return the text of the `read` method, with content type `text/plain` set on the RESPONSE.

`ZPythonScript_edit(params, body)`

Change the parameters and body of the script. This method accepts two arguments:

params
: The new value of the Python Script's parameters. Must be a comma seperated list of values in valid python function signature syntax. If it does not contain a valid signature string, a SyntaxError is raised.

body
: The new value of the Python Script's body. Must contain valid Python syntax. If it does not contain valid Python syntax, a SyntaxError is raised.

`ZPythonScript_setTitle(title)`

Change the script's title. This method accepts one argument, `title` which is the new value for the script's title and must be a string.

`ZPythonScriptHTML_upload(REQUEST, file="")`

Pass the text in file to the `write` method.

`write(text)`

Change the script by parsing the text argument into parts. Leading lines that begin with ## are stripped off, and if they are of the form ##`name=value`, they are used to set meta−data such as the title and parameters. The remainder of the text is set as the body of the Python Script.

`ZScriptHTML_tryParams()`

Return a list of the required parameters with which to test the script.

`read()`

Return the body of the Python Script, with a special comment block prepended. This block contains meta−data in the form of comment lines as expected by the `write` method.

`ZPythonScriptHTML_editAction(REQUEST, title, params, body)`

Change the script's main parameters. This method accepts the following arguments:

REQUEST
: The current request.

title
: The new value of the Python Script's title. This must be a string.

params
: The new value of the Python Script's parameters. Must be a comma seperated list of values in valid python function signature syntax. If it does not contain a valid signature string, a SyntaxError is raised.

body
: The new value of the Python Script's body. Must contain valid Python syntax. If it does not contain valid Python syntax, a SyntaxError is raised.

ObjectManager Constructor

`manage_addPythonScript(id, REQUEST=None)`

Add a Python script to a folder.

module `Request`

class `Request`

The request object encapsulates all of the information regarding the current request in Zope. This includes, the input headers, form data, server data, and cookies.

The request object is a mapping object that represents a collection of variable to value mappings. In addition, variables are divided into five categories:

- Environment variables

  These variables include input headers, server data, and other request−related data. The variable names are as specified in the CGI specification

- Form data

  These are data extracted from either a URL−encoded query string or body, if present.

- Cookies

  These are the cookie data, if present.

- Lazy Data

  These are callables which are deferred until explicitly referenced, at which point they are resolved (called) and the result stored as "other" data, ie regular request data.

  Thus, they are "lazy" data items. An example is SESSION objects.

  Lazy data in the request may only be set by the Python method set_lazy(name,callable) on the REQUEST object. This method is not callable from DTML or through the web.

- Other

  Data that may be set by an application object.

The request object may be used as a mapping object, in which case values will be looked up in the order: environment variables, other variables, form data, and then cookies.

These special variables are set in the Request:

`PARENTS`
: A list of the objects traversed to get to the published object. So, `PARENTS[0]` would be the ancestor of the published object.

`REQUEST`
: The Request object.

`RESPONSE`
: The Response object.

`PUBLISHED`
: The actual object published as a result of url traversal.

`URL`
: The URL of the Request without query string.

URLn
: `URL0` is the same as `URL`. `URL1` is the same as `URL0` with the last path element removed. `URL2` is the same as `URL1` with the last element removed. Etcetera.
For example if URL='http://localhost/foo/bar', then URL1='http://localhost/foo' and URL2='http://localhost'.

URLPATHn
: `URLPATH0` is the path portion of `URL`, `URLPATH1` is the path portion of `URL1`, and so on.
For example if URL='http://localhost/foo/bar', then URLPATH1='/foo' and URLPATH2='/'.

BASEn
: `BASE0` is the URL up to but not including the Zope application object. `BASE1` is the URL of the Zope application object. `BASE2` is the URL of the Zope application object with an additional path element added in the path to the published object. Etcetera.
For example if URL='http://localhost/Zope.cgi/foo/bar', then BASE0='http://localhost', BASE1='http://localhost/Zope.cgi', and BASE2='http://localhost/Zope.cgi/foo'.

BASEPATHn
: `BASEPATH0` is the path portion of `BASE0`, `BASEPATH1` is the path portion of `BASE1`, and so on. `BASEPATH1` is the externally visible path to the root Zope folder, equivalent to CGI's `SCRIPT_NAME`, but virtual−host aware.
For example if URL='http://localhost/Zope.cgi/foo/bar', then BASEPATH0='/', BASEPATH1='/Zope.cgi', and BASEPATH2='/Zope.cgi/foo'.

`get_header(name, default=None)`

Return the named HTTP header, or an optional default argument or None if the header is not found. Note that both original and CGI header names without the leading `HTTP_` are recognized, for example, `Content−Type`, `CONTENT_TYPE` and `HTTP_CONTENT_TYPE` should all return the Content−Type header, if available.

Permission
: Always available

`items()`

Returns a sequence of (key, value) tuples for all the keys in the REQUEST object.

Permission
: Always available

`keys()`

Returns a sorted sequence of all keys in the REQUEST object.

Permission
: Always available

`setVirtualRoot(path, hard=0)`

Alters `URL`, `URLn`, `URLPATHn`, `BASEn`, `BASEPATHn`, and `absolute_url()` so that the current object has path `path`. If `hard` is true, `PARENTS` is emptied.

Provides virtual hosting support. Intended to be called from publishing traversal hooks.

Permission
: Always available

`values()`

Returns a sequence of values for all the keys in the REQUEST object.

Permission
: Always available

`set(name, value)`

Create a new name in the REQUEST object and assign it a value. This name and value is stored in the `Other` category.

Permission
: Always available

`has_key(key)`

Returns a true value if the REQUEST object contains key, returns a false value otherwise.

Permission
: Always available

`setServerURL(protocol=None, hostname=None, port=None)`

Sets the specified elements of `SERVER_URL`, also affecting `URL`, `URLn`, `BASEn`, and `absolute_url()`.

Provides virtual hosting support.

Permission
: Always available

module Response

class Response

The Response object represents the response to a Zope request.

setHeader(name, value)

Sets an HTTP return header "name" with value "value", clearing the previous value set for the header, if one exists. If the literal flag is true, the case of the header name is preserved, otherwise word−capitalization will be performed on the header name on output.

Permission
: Always available

setCookie(name, value, **kw)

Set an HTTP cookie on the browser

The response will include an HTTP header that sets a cookie on cookie−enabled browsers with a key "name" and value "value". This overwrites any previously set value for the cookie in the Response object.

Permission
: Always available

addHeader(name, value)

Set a new HTTP return header with the given value, while retaining any previously set headers with the same name.

Permission
: Always available

appendHeader(name, value, delimiter=,)

Append a value to a cookie

Sets an HTTP return header "name" with value "value", appending it following a comma if there was a previous value set for the header.

Permission
: Always available

write(data)

Return data as a stream

HTML data may be returned using a stream−oriented interface. This allows the browser to display partial results while computation of a response to proceed.

The published object should first set any output headers or cookies on the response object.

Note that published objects must not generate any errors after beginning stream−oriented output.

Permission
    Always available

`setStatus(status, reason=None)`

Sets the HTTP status code of the response; the argument may either be an integer or one of the following strings:

OK, Created, Accepted, NoContent, MovedPermanently, MovedTemporarily, NotModified, BadRequest, Unauthorized, Forbidden, NotFound, InternalError, NotImplemented, BadGateway, ServiceUnavailable

that will be converted to the correct integer value.

Permission
    Always available

`setBase(base)`

Set the base URL for the returned document.

Permission
    Always available

`expireCookie(name, **kw)`

Cause an HTTP cookie to be removed from the browser

The response will include an HTTP header that will remove the cookie corresponding to "name" on the client, if one exists. This is accomplished by sending a new cookie with an expiration date that has already passed. Note that some clients require a path to be specified − this path must exactly match the path given when creating the cookie. The path can be specified as a keyword argument.

Permission
    Always available

`appendCookie(name, value)`

Returns an HTTP header that sets a cookie on cookie−enabled browsers with a key "name" and value "value". If a value for the cookie has previously been set in the response object, the new value is appended to the old one separated by a colon.

Permission
    Always available

`redirect(location, lock=0)`

Cause a redirection without raising an error. If the "lock" keyword argument is passed with a true value, then the HTTP redirect response code will not be changed even if an error occurs later in request processing (after redirect() has been called).

Permission
: Always available

module `Script`

class `Script`

Web−callable script base interface.

`ZScriptHTML_tryAction(REQUEST, argvars)`

Apply the test parameters provided by the dictionary `argvars`. This will call the current script with the given arguments and return the result.

module `SessionInterfaces`

Session API

See Also

- Transient Object API

class `SessionDataManagerErr`

Error raised during some session data manager operations, as explained in the API documentation of the Session Data Manager.

This exception may be caught in PythonScripts. A successful import of the exception for PythonScript use would need to be:

```
        from Products.Sessions import SessionDataManagerErr
```

class `BrowserIdManagerInterface`

Zope Browser Id Manager interface.

A Zope Browser Id Manager is responsible for assigning ids to site visitors, and for servicing requests from Session Data Managers related to the browser id.

`getBrowserId(self, create=1)`

If create=0, returns a the current browser id or None if there is no browser id associated with the current request. If create=1, returns the current browser id or a newly−created browser id if there is no browser id associated with the current request. This method is useful in conjunction with getBrowserIdName if you wish to embed the browser−id−name/browser−id combination as a hidden value in a POST−based form. The browser id is opaque, has no business meaning, and its length, type, and composition are subject to change.

Permission required: Access contents information

Raises: BrowserIdManagerErr if ill−formed browser id is found in REQUEST.

`isBrowserIdFromCookie(self)`

Returns true if browser id comes from a cookie.

Permission required: Access contents information

Raises: BrowserIdManagerErr. If there is no current browser id.

`isBrowserIdNew(self)`

Returns true if browser id is `new`. A browser id is `new` when it is first created and the client has therefore not sent it back to the server in any request.

Permission required: Access contents information

Raises: BrowserIdManagerErr. If there is no current browser id.

`encodeUrl(self, url)`

Encodes a provided URL with the current request's browser id and returns the result. For example, the call encodeUrl('http://foo.com/amethod') might return `http://foo.com/amethod?_ZopeId=as9dfu0adfu0ad`.

Permission required: Access contents information

Raises: BrowserIdManagerErr. If there is no current browser id.

`flushBrowserIdCookie(self)`

Deletes the browser id cookie from the client browser, iff the `cookies` browser id namespace is being used.

Permission required: Access contents information

Raises: BrowserIdManagerErr. If the `cookies` namespace isn't a browser id namespace at the time of the call.

`getBrowserIdName(self)`

Returns a string with the name of the cookie/form variable which is used by the current browser id manager as the name to look up when attempting to obtain the browser id value. For example, `_ZopeId`.

Permission required: Access contents information

`isBrowserIdFromForm(self)`

Returns true if browser id comes from a form variable (query string or post).

Permission required: Access contents information

Raises: BrowserIdManagerErr. If there is no current browser id.

`hasBrowserId(self)`

Returns true if there is a browser id for this request.

Permission required: Access contents information

`setBrowserIdCookieByForce(self, bid)`

Sets the browser id cookie to browser id `bid` by force. Useful when you need to `chain` browser id cookies across domains for the same user (perhaps temporarily using query strings).

Permission required: Access contents information

Raises: BrowserIdManagerErr. If the `cookies` namespace isn't a browser id namespace at the time of the call.

class `BrowserIdManagerErr`

Error raised during some browser id manager operations, as explained in the API documentation of the Browser Id Manager.

This exception may be caught in PythonScripts. A successful import of the exception for PythonScript use would need to be:

```
from Products.Sessions import BrowserIdManagerErr
```

class `SessionDataManagerInterface`

Zope Session Data Manager interface.

A Zope Session Data Manager is responsible for maintaining Session Data Objects, and for servicing requests from application code related to Session Data Objects. It also communicates with a Browser Id Manager to provide information about browser ids.

`getSessionDataByKey(self, key)`

Returns a Session Data Object associated with `key`. If there is no Session Data Object associated with `key` return None.

Permission required: Access arbitrary user session data

`getSessionData(self, create=1)`

Returns a Session Data Object associated with the current browser id. If there is no current browser id, and create is true, returns a new Session Data Object. If there is no current browser id and create is false, returns None.

Permission required: Access session data

`getBrowserIdManager(self)`

Returns the nearest acquirable browser id manager.

Raises SessionDataManagerErr if no browser id manager can be found.

Permission required: Access session data

`hasSessionData(self)`

Returns true if a Session Data Object associated with the current browser id is found in the Session Data Container. Does not create a Session Data Object if one does not exist.

Permission required: Access session data

module `TransienceInterfaces`

Transient Objects

class `TransientObject`

A transient object is a temporary object contained in a transient object container.

Most of the time you'll simply treat a transient object as a dictionary. You can use Python sub−item notation:

```
SESSION['foo']=1
foo=SESSION['foo']
del SESSION['foo']
```

When using a transient object from Python−based Scripts or DTML you can use the `get`, `set`, and `delete` methods instead.

Methods of transient objects are not protected by security assertions.

It's necessary to reassign mutuable sub−items when you change them. For example:

```
l=SESSION['myList']
l.append('spam')
SESSION['myList']=l
```

This is necessary in order to save your changes. Note that this caveat is true even for mutable subitems which inherit from the Persistence.Persistent class.

`delete(self, k)`

Call __delitem__ with key k.

Permission
    Always available

`setLastAccessed(self)`

Cause the last accessed time to be set to now.

Permission
: Always available

`getCreated(self)`

Return the time the transient object was created in integer seconds−since−the−epoch form.

Permission
: Always available

`values(self)`

Return sequence of value elements.

Permission
: Always available

`has_key(self, k)`

Return true if item referenced by key k exists.

Permission
: Always available

`getLastAccessed(self)`

Return the time the transient object was last accessed in integer seconds−since−the−epoch form.

Permission
: Always available

`getId(self)`

Returns a meaningful unique id for the object.

Permission
: Always available

`update(self, d)`

Merge dictionary d into ourselves.

Permission
: Always available

`clear(self)`

Remove all key/value pairs.

Permission
: Always available

`items(self)`

Return sequence of (key, value) elements.

Permission
: Always available

`keys(self)`

Return sequence of key elements.

Permission
: Always available

`get(self, k, default=marker)`

Return value associated with key k. If k does not exist and default is not marker, return default, else raise KeyError.

Permission
: Always available

`set(self, k, v)`

Call __setitem__ with key k, value v.

Permission
: Always available

`getContainerKey(self)`

Returns the key under which the object is "filed" in its container. getContainerKey will often return a differnt value than the value returned by getId.

Permission
: Always available

`invalidate(self)`

Invalidate (expire) the transient object.

Causes the transient object container's "before destruct" method related to this object to be called as a side effect.

Permission

Always available

class `MaxTransientObjectsExceeded`

An exception importable from the Products.Transience.Transience module which is raised when an attempt is made to add an item to a TransientObjectContainer that is `full`.

This exception may be caught in PythonScripts through a normal import. A successful import of the exception can be achieved via:

```
from Products.Transience import MaxTransientObjectsExceeded
```

class `TransientObjectContainer`

TransientObjectContainers hold transient objects, most often, session data.

You will rarely have to script a transient object container. You'll almost always deal with a TransientObject itself which you'll usually get as `REQUEST.SESSION`.

`new(self, k)`

Creates a new subobject of the type supported by this container with key "k" and returns it.

If an object already exists in the container with key "k", a KeyError is raised.

"k" must be a string, else a TypeError is raised.

If the container is `full`, a MaxTransientObjectsExceeded will be raised.

Permission
    `Create Transient Objects`

`setDelNotificationTarget(self, f)`

Cause the `before destruction` function to be `f`.

If `f` is not callable and is a string, treat it as a Zope path to a callable function.

`before destruction` functions need accept a single argument: `item`, which is the item being destroyed.

Permission
    `Manage Transient Object Container`

`getTimeoutMinutes(self)`

Return the number of minutes allowed for subobject inactivity before expiration.

Permission
    `View management screens`

`has_key(self, k)`

Return true if container has value associated with key k, else return false.

Permission
: `Access Transient Objects`

`setAddNotificationTarget(self, f)`

Cause the `after add` function to be `f`.

If `f` is not callable and is a string, treat it as a Zope path to a callable function.

`after add` functions need accept a single argument: `item`, which is the item being added to the container.

Permission
: `Manage Transient Object Container`

`getId(self)`

Returns a meaningful unique id for the object.

Permission
: Always available

`setTimeoutMinutes(self, timeout_mins)`

Set the number of minutes of inactivity allowable for subobjects before they expire.

Permission
: `Manage Transient Object Container`

`new_or_existing(self, k)`

If an object already exists in the container with key "k", it is returned.

Otherwiser, create a new subobject of the type supported by this container with key "k" and return it.

"k" must be a string, else a TypeError is raised.

If the container is `full`, a MaxTransientObjectsExceeded exception be raised.

Permission
: `Create Transient Objects`

`get(self, k, default=None)`

Return value associated with key k. If value associated with k does not exist, return default.

Permission
: `Access Transient Objects`

`getAddNotificationTarget(self)`

Returns the current `after add` function, or None.

Permission
    `View management screens`

`getDelNotificationTarget(self)`

Returns the current `before destruction` function, or None.

Permission
    `View management screens`

module `UserFolder`

class `UserFolder`

User Folder objects are containers for user objects. Programmers can work with collections of user objects using the API shared by User Folder implementations.

`userFolderEditUser(name, password, roles, domains, **kw)`

API method for changing user object attributes. Note that not all user folder implementations support changing of user object attributes. Implementations that do not support changing of user object attributes will raise an error for this method.

Permission
    Manage users

`userFolderDelUsers(names)`

API method for deleting one or more user objects. Note that not all user folder implementations support deletion of user objects. Implementations that do not support deletion of user objects will raise an error for this method.

Permission
    Manage users

`userFolderAddUser(name, password, roles, domains, **kw)`

API method for creating a new user object. Note that not all user folder implementations support dynamic creation of user objects. Implementations that do not support dynamic creation of user objects will raise an error for this method.

Permission
    Manage users

`getUsers()`

Returns a sequence of all user objects which reside in the user folder.

Permission
: Manage users

`getUserNames()`

Returns a sequence of names of the users which reside in the user folder.

Permission
: Manage users

`getUser(name)`

Returns the user object specified by name. If there is no user named `name` in the user folder, return None.

Permission
: Manage users

module `Vocabulary`

class `Vocabulary`

A Vocabulary manages words and language rules for text indexing. Text indexing is done by the ZCatalog and other third party Products.

`words()`

Return list of words.

`insert(word)`

Insert a word in the Vocabulary.

`query(pattern)`

Query Vocabulary for words matching pattern.

ObjectManager Constructor

`manage_addVocabulary(id, title, globbing=None, REQUEST=None)`

Add a Vocabulary object to an ObjectManager.

module `ZCatalog`

class `ZCatalog`

ZCatalog object

A ZCatalog contains arbitrary index like references to Zope objects. ZCatalog's can index either `Field` values of object, `Text` values, or `KeyWord` values:

ZCatalogs have three types of indexes:

Text
: Text indexes index textual content. The index can be used to search for objects containing certain words.

Field
: Field indexes index atomic values. The index can be used to search for objects that have certain properties.

Keyword
: Keyword indexes index sequences of values. The index can be used to search for objects that match one or more of the search terms.

The ZCatalog can maintain a table of extra data about cataloged objects. This information can be used on search result pages to show information about a search result.

The meta−data table schema is used to build the schema for ZCatalog Result objects. The objects have the same attributes as the column of the meta−data table.

ZCatalog does not store references to the objects themselves, but rather to a unique identifier that defines how to get to the object. In Zope, this unique identifier is the object's relative path to the ZCatalog (since two Zope objects cannot have the same URL, this is an excellent unique qualifier in Zope).

`schema()`

Returns a sequence of names that correspond to columns in the meta−data table.

`__call__(REQUEST=None, **kw)`

Search the catalog, the same way as `searchResults`.

`uncatalog_object(uid)`

Uncatalogs the object with the unique identifier `uid`.

`getobject(rid, REQUEST=None)`

Return a cataloged object given a `data_record_id_`

`indexes()`

Returns a sequence of names that correspond to indexes.

`getpath(rid)`

Return the path to a cataloged object given a `data_record_id_`

`index_objects()`

Returns a sequence of actual index objects.

`searchResults(REQUEST=None, **kw)`

Search the catalog. Search terms can be passed in the REQUEST or as keyword arguments.

Search queries consist of a mapping of index names to search parameters. You can either pass a mapping to searchResults as the variable `REQUEST` or you can use index names and search parameters as keyword arguments to the method, in other words:

```
searchResults(title='Elvis Exposed',
                 author='The Great Elvonso')
```

is the same as:

```
searchResults({'title' : 'Elvis Exposed',
                  'author : 'The Great Elvonso'})
```

In these examples, `title` and `author` are indexes. This query will return any objects that have the title *Elvis Exposed* AND also are authored by *The Great Elvonso*. Terms that are passed as keys and values in a searchResults() call are implicitly ANDed together. To OR two search results, call searchResults() twice and add concatenate the results like this:

```
results = ( searchResults(title='Elvis Exposed') +
               searchResults(author='The Great Elvonso') )
```

This will return all objects that have the specified title OR the specified author.

There are some special index names you can pass to change the behavior of the search query:

sort_on
: This parameters specifies which index to sort the results on.

sort_order
: You can specify `reverse` or `descending`. Default behavior is to sort ascending.

There are some rules to consider when querying this method:

- an empty query mapping (or a bogus REQUEST) returns all items in the catalog.
- results from a query involving only field/keyword indexes, e.g. {'id':'foo'} and no `sort_on` will be returned unsorted.
- results from a complex query involving a field/keyword index *and* a text index, e.g. {'id':'foo','PrincipiaSearchSource':'bar'} and no `sort_on` will be returned unsorted.
- results from a simple text index query e.g.{'PrincipiaSearchSource':'foo'} will be returned sorted in descending order by `score`. A text index cannot beused as a `sort_on` parameter, and attempting to do so will raise an error.

Depending on the type of index you are querying, you may be able to provide more advanced search parameters that can specify range searches or wildcards. These features are documented in The Zope Book.

`uniqueValuesFor(name)`

returns the unique values for a given FieldIndex named `name`.

`catalog_object(obj, uid)`

Catalogs the object `obj` with the unique identifier `uid`.

ObjectManager Constructor

`manage_addZCatalog(id, title, vocab_id=None)`

Add a ZCatalog object.

`vocab_id` is the name of a Vocabulary object this catalog should use. A value of None will cause the Catalog to create its own private vocabulary.

module `ZSQLMethod`

class `ZSQLMethod`

ZSQLMethods abstract SQL code in Zope.

SQL Methods behave like methods of the folders they are accessed in. In particular, they can be used from other methods, like Documents, ExternalMethods, and even other SQL Methods.

Database methods support the Searchable Object Interface. Search interface wizards can be used to build user interfaces to them. They can be used in joins and unions. They provide meta−data about their input parameters and result data.

For more information, see the searchable−object interface specification.

Database methods support URL traversal to access and invoke methods on individual record objects. For example, suppose you had an `employees` database method that took a single argument `employee_id`. Suppose that employees had a `service_record` method (defined in a record class or acquired from a folder). The `service_record` method could be accessed with a URL like:

```
employees/employee_id/1234/service_record
```

Search results are returned as Record objects. The schema of a Record objects matches the schema of the table queried in the search.

`manage_edit(title, connection_id, arguments, template)`

Change database method properties.

The `connection_id` argument is the id of a database connection that resides in the current folder or in a folder above the current folder. The database should understand SQL.

The `arguments` argument is a string containing an arguments specification, as would be given in the SQL method creation form.

The `template` argument is a string containing the source for the SQL Template.

`__call__(REQUEST=None, **kw)`

Call the ZSQLMethod.

The arguments to the method should be passed via keyword arguments, or in a single mapping object. If no arguments are given, and if the method was invoked through the Web, then the method will try to acquire and use the Web REQUEST object as the argument mapping.

The returned value is a sequence of record objects.

ObjectManager Constructor

`manage_addZSQLMethod(id, title, connection_id, arguments, template)`

Add an SQL Method to an ObjectManager.

The `connection_id` argument is the id of a database connection that resides in the current folder or in a folder above the current folder. The database should understand SQL.

The `arguments` argument is a string containing an arguments specification, as would be given in the SQL method cration form.

The `template` argument is a string containing the source for the SQL Template.

module `ZTUtils`

ZTUtils: Page Template Utilities

The classes in this module are available from Page Templates.

class `Batch`

Batch − a section of a large sequence.

You can use batches to break up large sequences (such as search results) over several pages.

Batches provide Page Templates with similar functions as those built−in to `<dtml-in>`.

You can access elements of a batch just as you access elements of a list. For example:

```
>>> b=Batch(range(100), 10)
>>> b[5]
4
>>> b[10]
IndexError: list index out of range
```

Batches have these public attributes:

start
: The first element number (counting from 1).

first
: The first element index (counting from 0). Note that this is that same as start − 1.

end

The last element number (counting from 1).

orphan
: The desired minimum batch size. This controls how sequences are split into batches. If a batch smaller than the orphan size would occur, then no split is performed, and a batch larger than the batch size results.

overlap
: The number of elements that overlap between batches.

length
: The actual length of the batch. Note that this can be different than size due to orphan settings.

size
: The desired size. Note that this can be different than the actual length of the batch due to orphan settings.

previous
: The previous batch or None if this is the first batch.

next
: The next batch or None if this is the last batch.

`__init__(self, sequence, size, start=0, end=0, orphan=0, overlap=0)`

Creates a new batch given a sequence and a desired batch size.

sequence
: The full sequence.

size
: The desired batch size.

start
: The index of the start of the batch (counting from 0).

end
: The index of the end of the batch (counting from 0).

orphan
: The desired minimum batch size. This controls how sequences are split into batches. If a batch smaller than the orphan size would occur, then no split is performed, and a batch larger than the batch size results.

overlap
: The number of elements that overlap between batches.

module `math`

math: Python `math` module

The `math` module provides trigonometric and other math functions. It is a standard Python module.

Since Zope 2.4 requires Python 2.1, make sure to consult the Python 2.1 documentation.

See Also

Python `math` module documentation at Python.org

module `random`

random: Python `random` module

The `random` module provides pseudo−random number functions. With it, you can generate random numbers and select random elements from sequences. This module is a standard Python module.

Since Zope 2.4 requires Python 2.1, make sure to consult the Python 2.1 documentation.

See Also

Python `random` module documentation at Python.org

module sequence

sequence: Sequence sorting module

This module provides a `sort` function for use with DTML, Page Templates, and Python−based Scripts.

`def sort(seq, sort)`

Sort the sequence *seq* of objects by the optional sort schema *sort*. *sort* is a sequence of tuples `(key, func, direction)` that describe the sort order.

key
: Attribute of the object to be sorted.

func
: Defines the compare function (optional). Allowed values:
  - *"cmp"*: Standard Python comparison function
  - *"nocase"*: Case−insensitive comparison
  - *"strcoll" or "locale"*: Locale−aware string comparison
  - *"strcoll_nocase" or "locale_nocase"*: Locale−aware case−insensitive string comparison
  - *other*: A specified, user−defined comparison function, should return 1, 0, −1.

direction
: defines the sort direction for the key (optional). (allowed values: "asc", "desc")

DTML Examples

Sort child object (using the `objectValues` method) by id (using the `getId` method), ignoring case:

```
<dtml-in expr="_.sequence.sort(objectValues(),
                               (('getId', 'nocase'),))">
  <dtml-var getId> <br>
</dtml-in>
```

Sort child objects by title (ignoring case) and date (from newest to oldest):

```
<dtml-in expr="_.sequence.sort(objectValues(),
                               (('title', 'nocase'),
                                ('bobobase_modification_time',
                                'cmp', 'desc')
                               ))">
  <dtml-var title> <dtml-var bobobase_modification_time> <br>
</dtml-in>
```

Page Template Examples

You can use the `sequence.sort` function in Python expressions to sort objects. Here's an example that mirrors the DTML example above:

```
<table tal:define="objects here/objectValues;
                   sort_on python:(('title', 'nocase', 'asc'),
                                   ('bobobase_modification_time', 'cmp', 'desc'));
                   sorted_objects python:sequence.sort(objects, sort_on)">
  <tr tal:repeat="item sorted_objects">
    <td tal:content="item/title">title</td>
    <td tal:content="item/bobobase_modification_time">
      modification date</td>
  </tr>
</table>
```

This example iterates over a sorted list of object, drawing a table row for each object. The objects are sorted by title and modification time.

See Also

Python cmp function

module `standard`

1. PythonScripts.standard: Utility functions and classes

   The functions and classes in this module are available from Python−based scripts, DTML, and Page Templates.

`def structured_text(s)`

Convert a string in structured−text format to HTML.

See Also

Structured−Text Rules

`def html_quote(s)`

Convert characters that have special meaning in HTML to HTML character entities.

See Also

Python `cgi` module _in_cgi_module.html `escape` function.

def url_quote_plus(s)

Like url_quote but also replace blank space characters with +. This is needed for building query strings in some cases.

See Also

Python urllib module url_quote_plus function.

def dollars_and_cents(number)

Show a numeric value with a dollar symbol and two decimal places.

def sql_quote(s)

Convert single quotes to pairs of single quotes. This is needed to safely include values in Standard Query Language (SQL) strings.

def whole_dollars(number)

Show a numeric value with a dollar symbol.

def url_quote(s)

Convert characters that have special meaning in URLS to HTML character entities using decimal values.

See Also

Python urllib module url_quote function.

class DTML

DTML − temporary, security−restricted DTML objects

__init__(source, **kw)

Create a DTML object with source text and keyword variables. The source text defines the DTML source content. The optinal keyword arguments define variables.

call(client=None, REQUEST={}, **kw)

Render the DTML.

To accomplish its task, DTML often needs to resolve various names into objects. For example, when the code <dtml−var spam> is executed, the DTML engine tries to resolve the name spam.

In order to resolve names, you must be pass a namespace to the DTML. This can be done several ways:

- By passing a client object − If the argument client is passed, then names are looked up as attributes on the argument.

- By passing a `REQUEST` mapping − If the argument `REQUEST` is passed, then names are looked up as items on the argument. If the object is not a mapping, an TypeError will be raised when a name lookup is attempted.
- By passing keyword arguments −− names and their values can be passed as keyword arguments to the Method.

The namespace given to a DTML object is the composite of these three methods. You can pass any number of them or none at all. Names will be looked up first in the keyword argument, next in the client and finally in the mapping.

`def thousand_commas(number)`

Insert commas every three digits to the left of a decimal point in values containing numbers. For example, the value, "12000 widgets" becomes "12,000 widgets".

`def newline_to_br(s)`

Convert newlines and carriage−return and newline combinations to break tags.

module `string`

string: Python `string` module

The `string` module provides string manipulation, conversion, and searching functions. It is a standard Python module.

Since Zope 2.4 requires Python 2.1, make sure to consult the Python 2.1 documentation.

See Also

Python `string` module documentation at Python.org

Appendix C: Zope Page Templates Reference

Zope Page Templates are an HTML/XML generation tool. This appendix is a reference to Zope Page Templates standards: Tag Attribute Language (TAL), TAL Expression Syntax (TALES), and Macro Expansion TAL (METAL).

TAL Overview

The *Template Attribute Language* (TAL) standard is an attribute language used to create dynamic templates. It allows elements of a document to be replaced, repeated, or omitted.

The statements of TAL are XML attributes from the TAL namespace. These attributes can be applied to an XML or HTML document in order to make it act as a template.

A ***TAL statement*** has a name (the attribute name) and a body (the attribute value). For example, an `content` statement might look like `tal:content="string:Hello"`. The element on which a statement is defined is its ***statement element***. Most TAL statements require expressions, but the syntax and semantics of these expressions are not part of TAL. TALES is recommended for this purpose.

TAL Namespace

The TAL namespace URI and recommended alias are currently defined as:

```
xmlns:tal="http://xml.zope.org/namespaces/tal"
```

This is not a URL, but merely a unique identifier. Do not expect a browser to resolve it successfully.

Zope does not require an XML namespace declaration when creating templates with a content−type of `text/html`. However, it does require an XML namespace declaration for all other content−types.

TAL Statements

These are the tal statements:

- tal:attributes − dynamically change element attributes.
- tal:define − define variables.
- tal:condition − test conditions.
- tal:content − replace the content of an element.
- tal:omit−tag − remove an element, leaving the content of the element.
- tal:on−error − handle errors.
- tal:repeat − repeat an element.
- tal:replace − replace the content of an element and remove the element leaving the content.

Expressions used in statements may return values of any type, although most statements will only accept strings, or will convert values into a string representation. The expression language must define a value named *nothing* that is not a string. In particular, this value is useful for deleting elements or attributes.

Order of Operations

When there is only one TAL statement per element, the order in which they are executed is simple. Starting with the root element, each element's statements are executed, then each of its child elements is visited, in order, to do the same.

Any combination of statements may appear on the same elements, except that the `content` and `replace` statements may not appear together.

When an element has multiple statements, they are executed in this order:

1. `define`
2. `condition`
3. `repeat`
4. `content` or `replace`
5. `attributes`
6. `omit−tag`

Since the `on−error` statement is only invoked when an error occurs, it does not appear in the list.

The reasoning behind this ordering goes like this: You often want to set up variables for use in other statements, so `define` comes first. The very next thing to do is decide whether this element will be included at all, so `condition` is next; since the condition may depend on variables you just set, it comes after `define`. It is valuable be able to replace various parts of an element with different values on each iteration of a repeat, so `repeat` is next. It makes no sense to replace attributes and then throw them away, so `attributes` is last. The remaining statements clash, because they each replace or edit the statement element.

See Also

TALES Overview

METAL Overview

tal:attributes

tal:define

tal:condition

tal:content

tal:omit−tag

tal:on−error

tal:repeat

tal:replace

attributes: Replace element attributes

Syntax

`tal:attributes` syntax:

```
argument             ::= attribute_statement [';' attribute_statement]*
attribute_statement  ::= attribute_name expression
attribute_name       ::= [namespace ':'] Name
namespace            ::= Name
```

Note: If you want to include a semi−colon (;) in an `expression`*, it must be escaped by doubling it (;;).*

Description

The `tal:attributes` statement replaces the value of an attribute (or creates an attribute) with a dynamic value. You can qualify an attribute name with a namespace prefix, for example `html:table`, if you are generating an XML document with multiple namespaces. The value of each expression is converted to a string, if necessary.

If the expression associated with an attribute assignment evaluates to *nothing*, then that attribute is deleted from the statement element. If the expression evaluates to *default*, then that attribute is left unchanged. Each attribute assignment is independent, so attributes may be assigned in the same statement in which some attributes are deleted and others are left alone.

If you use `tal:attributes` on an element with an active `tal:replace` command, the `tal:attributes` statement is ignored.

If you use `tal:attributes` on an element with a `tal:repeat` statement, the replacement is made on each repetition of the element, and the replacement expression is evaluated fresh for each repetition.

Examples

Replacing a link:

```
<a href="/sample/link.html"
 tal:attributes="href here/sub/absolute_url">
```

Replacing two attributes:

```
<textarea rows="80" cols="20"
 tal:attributes="rows request/rows;cols request/cols">
```

condition: Conditionally insert or remove an element

Syntax

`tal:condition` syntax:

```
argument ::= expression
```

Description

The `tal:condition` statement includes the statement element in the template only if the condition is met, and omits it otherwise. If its expression evaluates to a *true* value, then normal processing of the element continues, otherwise the statement element is immediately removed from the template. For these purposes, the value *nothing* is false, and *default* has the same effect as returning a true value.

Note: Zope considers missing variables, None, zero, empty strings, and empty sequences false; all other values are true.

Examples

Test a variable before inserting it (the first example tests for existence and truth, while the second only tests for existence):

```
<p tal:condition="request/message | nothing"
 tal:content="request/message">message goes here</p>

<p tal:condition="exists:request/message"
 tal:content="request/message">message goes here</p>
```

Test for alternate conditions:

```
<div tal:repeat="item python:range(10)">
<p tal:condition="repeat/item/even">Even</p>
<p tal:condition="repeat/item/odd">Odd</p>
</div>
```

content: Replace the content of an element

Syntax

`tal:content` syntax:

```
argument ::= (['text'] | 'structure') expression
```

Description

Rather than replacing an entire element, you can insert text or structure in place of its children with the `tal:content` statement. The statement argument is exactly like that of `tal:replace`, and is interpreted in the same fashion. If the expression evaluates to *nothing*, the statement element is left childless. If the expression evaluates to *default*, then the element's contents are unchanged.

Note: The default replacement behavior is `text`.

Examples

Inserting the user name:

```
<p tal:content="user/getUserName">Fred Farkas</p>
```

Inserting HTML/XML:

```
<p tal:content="structure here/getStory">marked <b>up</b>
content goes here.</p>
```

See Also

`tal:replace`

define: Define variables

Syntax

`tal:define` syntax:

```
argument       ::= define_scope [';' define_scope]*
define_scope   ::= (['local'] | 'global') define_var
define_var     ::= variable_name expression
variable_name  ::= Name
```

Note: If you want to include a semi−colon (;) in an `expression`*, it must be escaped by doubling it (;;).*

Description

The `tal:define` statement defines variables. You can define two different kinds of TAL variables: local and global. When you define a local variable in a statement element, you can only use that variable in that element and the elements it contains. If you redefine a local variable in a contained element, the new definition hides the outer element's definition within the inner element. When you define a global variables, you can use it in any element processed after the defining element. If you redefine a global variable, you replace its definition for the rest of the template.

Note: local variables are the default

If the expression associated with a variable evaluates to *nothing*, then that variable has the value *nothing*, and may be used as such in further expressions. Likewise, if the expression evaluates to *default*, then the variable has the value *default*, and may be used as such in further expressions.

Examples

Defining a global variable:

```
tal:define="global company_name string:Zope Corp, Inc."
```

Defining two variables, where the second depends on the first:

```
tal:define="mytitle template/title; tlen python:len(mytitle)"
```

omit−tag: Remove an element, leaving its contents

Syntax

`tal:omit-tag` syntax:

```
argument ::= [ expression ]
```

Description

The `tal:omit-tag` statement leaves the contents of a tag in place while omitting the surrounding start and end tag.

If its expression evaluates to a *false* value, then normal processing of the element continues and the tag is not omitted. If the expression evaluates to a *true* value, or there is no expression, the statement tag is replaced with its contents.

Zope treats empty strings, empty sequences, zero, None, *nothing*, and *default* at false. All other values are considered true.

Examples

Unconditionally omitting a tag:

```
<div tal:omit-tag="" comment="This tag will be removed">
  <i>...but this text will remain.</i>
</div>
```

Conditionally omitting a tag:

```
<b tal:omit-tag="not:bold">I may be bold.</b>
```

The above example will omit the `b` tag if the variable `bold` is false.

Creating ten paragraph tags, with no enclosing tag:

```
<span tal:repeat="n python:range(10)"
      tal:omit-tag="">
  <p tal:content="n">1</p>
</span>
```

on−error: Handle errors

Syntax

`tal:on−error` syntax:

```
argument ::= (['text'] | 'structure') expression
```

Description

The `tal:on−error` statement provides error handling for your template. When a TAL statement produces an error, the TAL interpreter searches for a `tal:on−error` statement on the same element, then on the enclosing element, and so forth. The first `tal:on−error` found is invoked. It is treated as a `tal:content` statement.

A local variable `error` is set. This variable has these attributes:

`type`
: the exception type

`value`
: the exception instance

`traceback`
: the traceback object

The simplest sort of `tal:on−error` statement has a literal error string or *nothing* for an expression. A more complex handler may call a script that examines the error and either emits error text or raises an exception to propagate the error outwards.

Examples

Simple error message:

```
<b tal:on-error="string: Username is not defined!"
 tal:content="here/getUsername">Ishmael</b>
```

Removing elements with errors:

```
<b tal:on-error="nothing"
   tal:content="here/getUsername">Ishmael</b>
```

Calling an error−handling script:

```
<div tal:on-error="structure here/errorScript">
  ...
</div>
```

Here's what the error−handling script might look like:

```
## Script (Python) "errHandler"
##bind namespace=_
##
error=_['error']
if error.type==ZeroDivisionError:
    return "<p>Can't divide by zero.</p>"
else
    return """<p>An error ocurred.</p>
              <p>Error type: %s</p>
              <p>Error value: %s</p>""" % (error.type,
                                           error.value)
```

See Also

Python Tutorial: Errors and Exceptions

Python Built−in Exceptions

repeat: Repeat an element

Syntax

`tal:repeat` syntax:

```
argument      ::= variable_name expression
variable_name ::= Name
```

Description

The `tal:repeat` statement replicates a sub−tree of your document once for each item in a sequence. The expression should evaluate to a sequence. If the sequence is empty, then the statement element is deleted, otherwise it is repeated for each value in the sequence. If the expression is *default*, then the element is left unchanged, and no new variables are defined.

The `variable_name` is used to define a local variable and a repeat variable. For each repetition, the local variable is set to the current sequence element, and the repeat variable is set to an iteration object.

Repeat Variables

You use repeat variables to access information about the current repetition (such as the repeat index). The repeat variable has the same name as the local variable, but is only accessible through the built−in variable named `repeat`.

The following information is available from the repeat variable:

- *index* − repetition number, starting from zero.
- *number* − repetition number, starting from one.
- *even* − true for even−indexed repetitions (0, 2, 4, ...).
- *odd* − true for odd−indexed repetitions (1, 3, 5, ...).
- *start* − true for the starting repetition (index 0).
- *end* − true for the ending, or final, repetition.
- *first* − true for the first item in a group − see note below
- *last* − true for the last item in a group − see note below
- *length* − length of the sequence, which will be the total number of repetitions.
- *letter* − repetition number as a lower−case letter: "a" − "z", "aa" − "az", "ba" − "bz", ..., "za" − "zz", "aaa" − "aaz", and so forth.
- *Letter* − upper−case version of *letter*.
- *roman* − repetition number as a lower−case roman numeral: "i", "ii", "iii", "iv", "v", etc.
- *Roman* − upper−case version of *roman*.

You can access the contents of the repeat variable using path expressions or Python expressions. In path expressions, you write a three−part path consisting of the name `repeat`, the statement variable's name, and the name of the information you want, for example, `repeat/item/start`. In Python expressions, you use normal dictionary notation to get the repeat variable, then attribute access to get the information, for example, "python:repeat['item'].start".

Note that `first` and `last` are intended for use with sorted sequences. They try to divide the sequence into group of items with the same value. If you provide a path, then the value obtained by following that path from a sequence item is used for grouping, otherwise the value of the item is used. You can provide the path by passing it as a parameter, as in "python:repeat['item'].first('color')", or by appending it to the path from the repeat variable, as in "repeat/item/first/color".

Examples

Iterating over a sequence of strings::

Inserting a sequence of table rows, and using the repeat variable to number the rows:

```
<table>
  <tr tal:repeat="item here/cart">
      <td tal:content="repeat/item/number">1</td>
      <td tal:content="item/description">Widget</td>
      <td tal:content="item/price">$1.50</td>
  </tr>
</table>
```

Nested repeats:

```
<table border="1">
  <tr tal:repeat="row python:range(10)">
    <td tal:repeat="column python:range(10)">
      <span tal:define="x repeat/row/number;
                        y repeat/column/number;
                        z python:x*y"
            tal:replace="string:$x * $y = $z">1 * 1 = 1</span>
    </td>
  </tr>
</table>
```

Insert objects. Seperates groups of objects by meta−type by drawing a rule between them:

```
<div tal:repeat="object objects">
  <h2 tal:condition="repeat/object/first/meta_type"
    tal:content="object/meta_type">Meta Type</h2>
  <p tal:content="object/getId">Object ID</p>
  <hr tal:condition="repeat/object/last/meta_type" />
</div>
```

Note, the objects in the above example should already be sorted by meta−type.

replace: Replace an element

Syntax

`tal:replace` syntax:

```
argument ::= (['text'] | 'structure') expression
```

Description

The `tal:replace` statement replaces an element with dynamic content. It replaces the statement element with either text or a structure (unescaped markup). The body of the statement is an expression with an optional type prefix. The value of the expression is converted into an escaped string if you prefix the expression with `text` or omit the prefix, and is inserted unchanged if you prefix it with `structure`. Escaping consists of converting "&" to "&amp;", "<" to "&lt;", and ">" to "&gt;".

If the value is *nothing*, then the element is simply removed. If the value is *default*, then the element is left unchanged.

Examples

The two ways to insert the title of a template:

```
<span tal:replace="template/title">Title</span>
<span tal:replace="text template/title">Title</span>
```

Inserting HTML/XML:

```
<div tal:replace="structure table" />
```

Inserting nothing:

```
<div tal:replace="nothing">This element is a comment.</div>
```

See Also

`tal:content`

TALES Overview

The *Template Attribute Language Expression Syntax* (TALES) standard describes expressions that supply TAL and METAL with data. TALES is *one* possible expression syntax for these languages, but they are not bound to this definition. Similarly, TALES could be used in a context having nothing to do with TAL or METAL.

TALES expressions are described below with any delimiter or quote markup from higher language layers removed. Here is the basic definition of TALES syntax:

```
Expression  ::= [type_prefix ':'] String
type_prefix ::= Name
```

Here are some simple examples:

```
a/b/c
path:a/b/c
nothing
path:nothing
python: 1 + 2
string:Hello, ${user/getUserName}
```

The optional *type prefix* determines the semantics and syntax of the *expression string* that follows it. A given implementation of TALES can define any number of expression types, with whatever syntax you like. It also determines which expression type is indicated by omitting the prefix.

If you do not specify a prefix, Zope assumes that the expression is a *path* expression.

TALES Expression Types

These are the TALES expression types supported by Zope:

- path expressions − locate a value by its path.
- exists expressions − test whether a path is valid.
- nocall expressions − locate an object by its path.
- not expressions − negate an expression
- string expressions − format a string
- python expressions − execute a Python expression

Built−in Names

These are the names that always available to TALES expressions in Zope:

- *nothing* − special value used by to represent a *non−value* (e.g. void, None, Nil, NULL).
- *default* − special value used to specify that existing text should not be replaced. See the documentation for individual TAL statements for details on how they interpret *default*.

- *options* − the *keyword* arguments passed to the template. These are generally available when a template is called from Methods and Scripts, rather than from the web.
- *repeat* − the `repeat` variables; see the tal:repeat documentation.
- *attrs* − a dictionary containing the initial values of the attributes of the current statement tag.
- *CONTEXTS* − the list of standard names (this list). This can be used to access a built−in variable that has been hidden by a local or global variable with the same name.
- *root* − the system's top−most object: the Zope root folder.
- *here* − the object to which the template is being applied.
- *container* − The folder in which the template is located.
- *template* − the template itself.
- *request* − the publishing request object.
- *user* − the authenticated user object.
- *modules* − a collection through which Python modules and packages can be accessed. Only modules which are approved by the Zope security policy can be accessed.

Note the names `root`, `here`, `container`, `template`, `request`, `user`, and `modules` are optional names supported by Zope, but are not required by the TALES standard.

See Also

TAL Overview

METAL Overview

exists expressions

nocall expressions

not expressions

string expressions

path expressions

python expressions

TALES Exists expressions

Syntax

Exists expression syntax:

```
        exists_expressions ::= 'exists:' path_expression
```

Description

Exists expressions test for the existence of paths. An exists expression returns true when the path expressions following it expression returns a value. It is false when the path expression cannot locate an object.

Examples

Testing for the existence of a form variable:

```
<p tal:condition="not:exists:request/form/number">
  Please enter a number between 0 and 5
</p>
```

Note that in this case you can't use the expression, `not:request/form/number`, since that expression will be true if the `number` variable exists and is zero.

TALES Nocall expressions

Syntax

Nocall expression syntax:

```
nocall_expression ::= 'nocall:' path_expression
```

Description

Nocall expressions avoid rendering the results of a path expression.

An ordinary path expression tries to render the object that it fetches. This means that if the object is a function, Script, Method, or some other kind of executable thing, then expression will evaluate to the result of calling the object. This is usually what you want, but not always. For example, if you want to put a DTML Document into a variable so that you can refer to its properties, you can't use a normal path expression because it will render the Document into a string.

Examples

Using nocall to get the properties of a document:

```
<span tal:define="doc nocall:here/aDoc"
      tal:content="string:${doc/getId}: ${doc/title}">
Id: Title</span>
```

Using nocall expressions on a functions:

```
<p tal:define="join nocall:modules/string/join">
```

This example defines a variable `join` which is bound to the `string.join` function.

TALES Not expressions

Syntax

Not expression syntax:

```
not_expression ::= 'not:' expression
```

Description

Not expression evaluate the expression string (recursively) as a full expression, and returns the boolean negation of its value. If the expression supplied does not evaluate to a boolean value, *not* will issue a warning and *coerce* the expression's value into a boolean type based on the following rules:

1. the number 0 is *false*
2. numbers > 0 are *true*
3. an empty string or other sequence is *false*
4. a non−empty string or other sequence is *true*
5. a *non−value* (e.g. void, None, Nil, NULL, etc) is *false*
6. all other values are implementation−dependent.

If no expression string is supplied, an error should be generated.

Zope considers all objects not specifically listed above as *false* (including negative numbers) to be *true*.

Examples

Testing a sequence:

```
<p tal:condition="not:here/objectIds">
  There are no contained objects.
</p>
```

TALES Path expressions

Syntax

Path expression syntax:

```
PathExpr  ::= Path [ '|' Path ]*
Path      ::= variable [ '/' URL_Segment ]*
variable  ::= Name
```

Description

A path expression consists of one or more *paths* separated by vertical bars (|). A path consists of one or more non−empty strings separated by slashes. The first string must be a variable name (built−in variable or a user defined variable), and the remaining strings, the *path segments*, may contain letters, digits, spaces, and the punctuation characters underscore, dash, period, comma, and tilde.

For example:

```
request/cookies/oatmeal
nothing
here/some−file 2001_02.html.tar.gz/foo
root/to/branch | default
request/name | string:Anonymous Coward
```

When a path expression is evaluated, Zope attempts to traverse the path, from left to right, until it succeeds or runs out of paths segments. To traverse a path, it first fetches the object stored in the variable. For each path segment, it traverses from the current object to the subobject named by the path segment. Subobjects are located according to standard Zope traversal rules (via getattr, getitem, or traversal hooks).

Once a path has been successfully traversed, the resulting object is the value of the expression. If it is a callable object, such as a method or template, it is called.

If a traversal step fails, evaluation immediately proceeds to the next path. If there are no further paths, an error results.

The expression in a series of paths seperated by vertical bars can be any TALES expression. For example, 'request/name | string:Anonymous Coward'. This is useful chiefly for providing default values such as strings and numbers which are not expressable as path expressions.

If no path is given the result is *nothing*.

Since every path must start with a variable name, you need a set of starting variables that you can use to find other objects and values. See the TALES overview for a list of built−in variables. Since variable names are looked up first in locals, then in globals, then in this list, these names act just like built−ins in Python; They are always available, but they can be shadowed by a global or local variable declaration. You can always access the built−in names explicitly by prefixing them with *CONTEXTS*. (e.g. CONTEXTS/root, CONTEXTS/nothing, etc).

Examples

Inserting a cookie variable or a property:

```
<span tal:replace="request/cookies/pref | here/pref">
  preference
</span>
```

Inserting the user name:

```
<p tal:content="user/getUserName">
  User name
</p>
```

TALES Python expressions

Syntax

Python expression syntax:

```
Any valid Python language expression
```

Description

Python expressions evaluate Python code in a security−restricted environment. Python expressions offer the same facilities as those available in Python−based Scripts and DTML variable expressions.

Security Restrictions

Python expressions are subject to the same security restrictions as Python−based scripts. These restrictions include:

access limits
: Python expressions are subject to Zope permission and role security restrictions. In addition, expressions cannot access objects whose names begin with underscore.

write limits
: Python expressions cannot change attributes of Zope objects.

Despite these limits malicious Python expressions can cause problems. See The Zope Book for more information.

Built−in Functions

Python expressions have the same built−ins as Python−based Scripts with a few additions.

These standard Python built−ins are available: `None`, `abs`, `apply`, `callable`, `chr`, `cmp`, `complex`, `delattr`, `divmod`, `filter`, `float`, `getattr`, `hash`, `hex`, `int`, `isinstance`, `issubclass`, `list`, `len`, `long`, `map`, `max`, `min`, `oct`, `ord`, `repr`, `round`, `setattr`, `str`, `tuple`.

The `range` and `pow` functions are available and work the same way they do in standard Python; however, they are limited to keep them from generating very large numbers and sequences. This limitation helps protect against denial of service attacks.

In addition, these utility functions are available: `DateTime`, `test`, and `same_type`. See DTML functions for more information on these functions.

Finally, these functions are available in Python expressions, but not in Python−based scripts:

`path(string)`
: Evaluate a TALES path expression.

`string(string)`
: Evaluate a TALES string expression.

`exists(string)`
: Evaluates a TALES exists expression.

`nocall(string)`
: Evaluates a TALES nocall expression.

Python Modules

A number of Python modules are available by default. You can make more modules available. You can access modules either via path expressions (for example 'modules/string/join') or in Python with the `modules` mapping object (for example 'modules["string"].join'). Here are the default modules:

`string`
: The standard Python string module. Note: most of the functions in the module are also available as methods on string objects.

`random`
: The standard Python random module.

`math`
: The standard Python math module.

`sequence`
: A module with a powerful sorting function. See sequence for more information.

`Products.PythonScripts.standard`
: Various HTML formatting functions available in DTML. See Products.PythonScripts.standard for

more information.

ZTUtils
: Batch processing facilities similar to those offered by `dtml-in`. See ZTUtils for more information.

AccessControl
: Security and access checking facilities. See AccessControl for more information.

Examples

Using a module usage (pick a random choice from a list):

```
<span tal:replace="python:modules['random'].choice(['one',
                     'two', 'three', 'four', 'five'])">
  a random number between one and five
</span>
```

String processing (capitalize the user name):

```
<p tal:content="python:user.getUserName().capitalize()">
  User Name
</p>
```

Basic math (convert an image size to megabytes):

```
<p tal:content="python:image.getSize() / 1048576.0">
  12.2323
</p>
```

String formatting (format a float to two decimal places):

```
<p tal:content="python:'%0.2f' % size">
  13.56
</p>
```

TALES String expressions

Syntax

String expression syntax:

```
string_expression ::= ( plain_string | [ varsub ] )*
varsub            ::= ( '$' Path ) | ( '${' Path '}' )
plain_string      ::= ( '$$' | non_dollar )*
non_dollar        ::= any character except '$'
```

Description

String expressions interpret the expression string as text. If no expression string is supplied the resulting string is *empty*. The string can contain variable substitutions of the form `$name` or `${path}`, where `name` is a variable name, and `path` is a path expression. The escaped string value of the path expression is inserted into the string. To prevent a `$` from being interpreted this way, it must be escaped as `$$`.

Examples

Basic string formatting:

```
<span tal:replace="string:$this and $that">
  Spam and Eggs
</span>
```

Using paths:

```
<p tal:content="total: ${request/form/total}">
  total: 12
</p>
```

Including a dollar sign:

```
<p tal:content="cost: $$$cost">
  cost: $42.00
</p>
```

METAL Overview

The *Macro Expansion Template Attribute Language* (METAL) standard is a facility for HTML/XML macro preprocessing. It can be used in conjunction with or independently of TAL and TALES.

Macros provide a way to define a chunk of presentation in one template, and share it in others, so that changes to the macro are immediately reflected in all of the places that share it. Additionally, macros are always fully expanded, even in a template's source text, so that the template appears very similar to its final rendering.

METAL Namespace

The METAL namespace URI and recommended alias are currently defined as:

```
xmlns:metal="http://xml.zope.org/namespaces/metal"
```

Just like the TAL namespace URI, this URI is not attached to a web page; it's just a unique identifier.

Zope does not require an XML namespace declaration when creating templates with a content−type of `text/html`. However, it does require an XML namespace declaration for all other content−types.

METAL Statements

METAL defines a number of statements:

- metal:define−macro − Define a macro.
- metal:use−macro − Use a macro.
- metal:define−slot − Define a macro customization point.
- metal:fill−slot − Customize a macro.

Although METAL does not define the syntax of expression non−terminals, leaving that up to the implementation, a canonical expression syntax for use in METAL arguments is described in TALES Specification.

See Also

TAL Overview

TALES Overview

metal:define−macro

metal:use−macro

metal:define−slot

metal:fill−slot

define−macro: Define a macro

Syntax

`metal:define−macro` syntax:

```
argument ::= Name
```

Description

The `metal:define−macro` statement defines a macro. The macro is named by the statement expression, and is defined as the element and its sub−tree.

In Zope, a macro definition is available as a sub−object of a template's `macros` object. For example, to access a macro named `header` in a template named `master.html`, you could use the path expression `master.html/macros/header`.

Examples

Simple macro definition:

```
<p metal:define-macro="copyright">
  Copyright 2001, <em>Foobar</em> Inc.
</p>
```

See Also

metal:use−macro

metal:define−slot

define−slot: Define a macro customization point

Syntax

`metal:define−slot` syntax:

```
argument ::= Name
```

Description

The `metal:define-slot` statement defines a macro customization point or *slot*. When a macro is used, its slots can be replaced, in order to customize the macro. Slot definitions provide default content for the slot. You will get the default slot contents if you decide not to customize the macro when using it.

The `metal:define-slot` statement must be used inside a `metal:define-macro` statement.

Slot names must be unique within a macro.

Examples

Simple macro with slot:

```
<p metal:define-macro="hello">
  Hello <b metal:define-slot="name">World</b>
</p>
```

This example defines a macro with one slot named `name`. When you use this macro you can customize the `b` element by filling the `name` slot.

See Also

metal:fill−slot

fill−slot: Customize a macro

Syntax

`metal:fill-slot` syntax:

```
argument ::= Name
```

Description

The `metal:fill-slot` statement customizes a macro by replacing a *slot* in the macro with the statement element (and its content).

The `metal:fill-slot` statement must be used inside a `metal:use-macro` statement.

Slot names must be unique within a macro.

If the named slot does not exist within the macro, the slot contents will be silently dropped.

Examples

Given this macro:

```
<p metal:define-macro="hello">
  Hello <b metal:define-slot="name">World</b>
</p>
```

You can fill the `name` slot like so:

```
<p metal:use−macro="container/master.html/macros/hello">
  Hello <b metal:fill−slot="name">Kevin Bacon</b>
</p>
```

See Also

metal:define−slot

use−macro: Use a macro

Syntax

`metal:use−macro` syntax:

```
argument ::= expression
```

Description

The `metal:use−macro` statement replaces the statement element with a macro. The statement expression describes a macro definition.

In Zope the expression will generally be a path expression referring to a macro defined in another template. See "metal:define−macro" for more information.

The effect of expanding a macro is to graft a subtree from another document (or from elsewhere in the current document) in place of the statement element, replacing the existing sub−tree. Parts of the original subtree may remain, grafted onto the new subtree, if the macro has *slots*. See metal:define−slot for more information. If the macro body uses any macros, they are expanded first.

When a macro is expanded, its `metal:define−macro` attribute is replaced with the `metal:use−macro` attribute from the statement element. This makes the root of the expanded macro a valid `use−macro` statement element.

Examples

Basic macro usage:

```
<p metal:use−macro="container/other.html/macros/header">
  header macro from defined in other.html template
</p>
```

This example refers to the `header` macro defined in the `other.html` template which is in the same folder as the current template. When the macro is expanded, the `p` element and its contents will be replaced by the macro. Note: there will still be a `metal:use−macro` attribute on the replacement element.

See Also

metal:define−macro

metal:fill−slot

Appendix D: Zope Resources

At the time of this writing there is a multitude of sources for Zope information on the Internet, but very little in print. We've collected a number of the most important links which you can use to find out more about Zope.

Zope Web Sites

Zope.org is the official Zope web site. It has downloads, documentation, news, and lots of community resources.

ZopeZen is a Zope community site that features news and a Zope job board. The site is run by noted Zope community member Andy McKay.

Zope Newbies is a weblog that features Zope news and related information. Zope Newbies is one of the oldest and best Zope web sites. Jeff Shelton started Zope Newbies, and the site is currently run by Luke Tymowski.

Zope Documentation

Zope.org has lots of documentation including official documentation projects and contributed community documentation.

Zope Documentation Project is a community−run Zope documentation web site. It hosts original documentation and has links to other sources of documentation.

Zope Developer's Guide teaches you how to write Zope products.

Mailing Lists

Zope.org maintains a collection of the many Zope mailing lists.

Zope Extensions

Zope.org has a huge collection of 3rd party Zope extensions which are called "products".

Zope Treasures is a large collection of Zope products.

Python Information

Python.org has lots of information about Python including a tutorial and reference documentation.