

The Zope Book (2.6 Edition)

Amos Latteier, Michel Pelletier, Chris McDonough, Peter Sabaini

The Zope Book (2.6 Edition)

Preface	32
How the Book Is Organized	32
Conventions Used in This Book	34
Contributors to This Book	35
Introducing Zope	36
What Is A Web Application?	36
How You Can Benefit From Using An Application Server	37
Zope History	38
Why Use Zope Instead of Another Application Server	38
Zope Audiences and What Zope Isn't	39
Zope's Terms of Use and License and an Introduction to The Zope Community	40
Zope Concepts and Architecture	41
Fundamental Zope Concepts	41
Zope Is A Framework	41
Object Orientation	41
Object Publishing	41
Through-The-Web Management	42
Security and Safe Delegation	42
Native Object Persistence and Transactions	43
Acquisition	43
Zope Is Extensible	43
Fundamental Zope Components	44
Installing and Starting Zope	45
Downloading Zope	45
Installing Zope	45
Installing Zope for Windows With Binaries from Zope.org	46
Installing Zope on Linux and Solaris With Binaries from Zope.org	50
Compiling and Installing Zope from Source Code	52
Starting Zope	53
Using Zope With An Existing Webserver	54
Starting Zope On Windows	54
Starting Zope on UNIX	54
Starting Zope As The Root User	55
Your Zope Installation	55
Logging In	56
Controlling the Zope Process With the Control Panel	57

The Zope Book (2.6 Edition)

Controlling the Zope Process From the Command Line	57
Troubleshooting	58
Options To The Zope start or start.bat Script	58
Environment Variables that Effect Zope at Runtime	61
When All Else Fails	65
Object Orientation	66
Objects	66
Attributes	67
Methods	67
Messages	67
Classes and Instances	68
Inheritance	68
Object Lifetimes	69
Summary	69
Using The Zope Management Interface	70
Introduction	70
How The Zope Management Interface Relates to Objects	70
ZMI Frames	70
The Navigator Frame	70
The Workspace Frame	71
The Status Frame	72
Creating Objects	72
Moving and Renaming Objects	73
Transactions and Undoing Mistakes	75
Undo Details and Gotchas	76
Reviewing Change History	76
Importing and Exporting Objects	77
Using Object Properties	80
Using the Help System	82
Browsing and Searching Help	82
Logging Out	83
Using Basic Zope Objects	84
Basic Zope Objects	84
Content Objects: Folders, Files, and Images	84
Folders	84
Files	84
Creating and Editing Files	85

The Zope Book (2.6 Edition)

Editing File Contents	86
Viewing Files	86
Images	87
Presentation Objects: Zope Page Templates and DTML Objects	87
ZPT vs. DTML: Same Purpose, Different Audiences	88
Zope Page Templates	89
Creating A Page Template	89
Editing A Page Template	89
Uploading A Page Template	89
Viewing A Page Template	90
DTML Objects: DTML Documents and DTML Methods	90
Creating DTML Methods	91
Editing DTML Methods	91
Viewing a DTML Method	92
Uploading an HTML File as Content for a DTML Method	92
Logic Objects: Script (Python) Objects and External Methods	93
Script (Python) Objects	93
Creating A Script (Python)	94
Editing A Script (Python)	94
Testing A Script (Python)	94
Uploading A Script (Python)	95
External Methods	96
Creating and Editing An External Method File	96
Creating an External Method Object	96
Testing An External Method Object	96
SQL Methods: Another Kind of Logic Object	97
Creating a Basic Zope Application Using Page Templates and Scripts	97
Creating a Data Collection Form	98
Creatng A Script To Calculate Interest Rates	98
Creating A Page Template To Display Results	99
Dealing With Errors	99
Using The Application	100
The Zope Tutorial	100
Acquisition	102
Acquisition vs. Inheritance	102
Acquisition is about Containment	103
Say What?	103

Providing Services	104
Getting Deeper with Multiple Levels	104
Summary	104
Basic DTML	106
How DTML Relates to Similar Languages and Templating Facilities	106
When To Use DTML	106
When Not To Use DTML	106
The Difference Between DTML Documents and DTML Methods	107
Details	107
DTML Tag Syntax	108
DTML Tag Names, Targets, and Attributes	108
Creating a "Sandbox" for the Examples in This Chapter	109
Examples of Using DTML for Common Tasks	109
Inserting Text into HTML with DTML	109
Formatting and Displaying Sequences	111
Processing Input from Forms	112
Dealing With Errors	115
Dynamically Acquiring Content	115
Using Python Expressions from DTML	117
DTML Expression Gotchas	119
will call the method. However,	119
Common DTML Tags	120
The Var Tag	120
Var Tag Attributes	120
Var Tag Entity Syntax	121
The If Tag	121
Here's an example condition:	121
Name and Expression Syntax Differences	122
Else and Elif Tags	122
Using Cookies with the If Tag	123
The In Tag	124
Iterating over Folder Contents	124
In Tag Special Variables	125
Summary	127
Using Zope Page Templates	128
Zope Page Templates versus DTML	128
How Page Templates Work	128

The Zope Book (2.6 Edition)

Creating a Page Template	129
Simple Expressions	130
Inserting Text	130
Repeating Structures	131
Conditional Elements	132
Changing Attributes	133
Creating a File Library with Page Templates	133
Remote Editing with FTP and WebDAV	136
Debugging and Testing	137
XML Templates	138
Using Templates with Content	138
Creating Basic Zope Applications	140
Building "Instance-Space" Applications	140
Instance-Space Applications vs. Products	140
Using A Folder as A Container For Your Instance-Space Application	140
Using Objects as Methods Of Folders Via URLs	141
Using Acquisition In Instance-Space Applications	141
The Special Folder Object index_html	141
Building the Zope Zoo Website	142
Navigating the Zoo	142
Adding a Front Page to the Zoo	144
Improving Navigation	145
Factoring out Style Sheets	147
Creating a File Library	148
	148
Building a Guest Book	150
Extending the Guest Book to Generate XML	153
The Next Step	154
Users and Security	155
Introduction to Zope Security	155
Review: Logging In and Logging Out of the Zope Management Interface	155
Zope's "Stock" Security Setup	155
Identification and Authentication	156
Authorization, Roles, and Permissions	156
Managing Users	157
Creating Users in User Folders	157
Editing Users	159

The Zope Book (2.6 Edition)

Defining a User's Location	159
Working with Alternative User Folders	160
Special User Accounts	160
Zope Anonymous User	161
Zope Emergency User	161
Creating an Emergency User	162
Zope Initial Manager	163
Protecting Against Password Snooping	163
Managing Custom Security Policies	164
Working with Roles	164
Defining Global Roles	164
Understanding Local Roles	165
Understanding Permissions	165
Defining Security Policies	166
Security Policy Acquisition	167
Security Usage Patterns	168
Security Rules of Thumb	168
Global and Local Policies	168
Delegating Control to Local Managers	168
Different Levels of Access with Roles	169
Controlling Access to Locations with Roles	170
Performing Security Checks	170
Advanced Security Issues: Ownership and Executable Content	172
The Problem: Trojan Horse Attacks	172
Managing Ownership	172
Roles of Executable Content	173
Proxy Roles	174
Summary	174
Advanced DTML	176
How Variables are Looked up	177
DTML Namespaces	178
DTML Client Object	179
DTML Method vs. DTML Document	180
DTML Request Object	180
Rendering Variables	181
Modifying the DTML Namespace	181
In Tag Namespace Modifications	181

The Zope Book (2.6 Edition)

Additional Notes	182
The With Tag	182
The Let Tag	183
DTML Namespace Utility Functions	184
DTML Security	185
Safe Scripting Limits	186
Advanced DTML Tags	186
The Call Tag	186
The Comment Tag	187
The Tree Tag	188
The Return Tag	190
The Sendmail Tag	190
The Mime Tag	191
The Unless Tag	192
Batch Processing With The In Tag	193
Exception Handling Tags	195
The Raise Tag	195
The Try Tag	195
The Try Tag Optional Else Block	197
The Try Tag Optional Finally Block	197
Other useful examples	198
Forwarding a REQUEST	198
Sorting with the tag	198
Calling a DTML object from a Python Script	199
Explicit Lookups	199
Conclusion	199
Advanced Page Templates	200
Advanced TAL	200
Advanced Content Insertion	200
Inserting Structure	200
Dummy Elements	200
Default Content	201
Advanced Repetition	201
Repeat Variables	201
Repetition Tips	202
Advanced Attribute Control	203
Defining Variables	203

The Zope Book (2.6 Edition)

Omitting Tags	204
Error Handling	204
Interactions Between TAL Statements	205
Form Processing	207
Expressions	208
Built-in Page Template Variables	208
String Expressions	209
Path Expressions	210
Alternate Paths	210
Not Expressions	211
Nocall Expressions	211
Exists Expressions	211
Python Expressions	212
Comparisons	212
Using other Expression Types	212
Getting at Zope Objects	213
Using Scripts	214
Calling DTML	214
Python Modules	215
Macros	215
Using Macros	216
Macro Details	216
Using Slots	217
Customizing Default Presentation	218
Combining METAL and TAL	219
Whole Page Macros	219
Caching Templates	220
Page Template Utilities	221
Batching Large Sets of Information	221
Miscellaneous Utilities	223
Conclusion	223
Advanced Zope Scripting	224
Zope Scripts	224
Here is an overview of Zope's scripts:	224
Calling Scripts	224
Context	225
Calling Scripts From the Web	225

The Zope Book (2.6 Edition)

URL Traversal and Acquisition	226
Passing Arguments with an HTTP Query String	226
Calling Scripts from Other Objects	226
Calling Scripts from DTML	226
Calling scripts from Python and Perl	227
Calling Scripts from Page Templates	228
Calling Scripts: Summary and Comparison	229
Using Python-based Scripts	230
The Python Language	230
Creating Python-based Scripts	230
Binding Variables	232
Accessing the HTTP Request	233
String Processing in Python	234
Doing Math	234
Print Statement Support	235
Built-in Functions	236
Using External Methods	236
Processing XML with External Methods	241
External Method Gotchas	242
Using Perl-based Scripts	242
The Perl Language	243
Creating Perl-based Scripts	243
Perl-based Script Security	244
Advanced Acquisition	244
Context Acquisition Gotchas	246
Containment before context	246
One at a time	246
Readability	246
Fragility	247
Calling DTML from Scripts	247
Calling ZPT from Scripts	248
Passing Parameters to Scripts	249
Returning Values from Scripts	253
Script Security	254
Security Restrictions of Script (Python)	254
The Zope API	255
Get all objects in a folder	255

The Zope Book (2.6 Edition)

Get the id of an object	256
Get the Zope root folder	256
Get the physical path to an object	256
Get an object by path	256
Change the content of an DTML Method or Document	256
Change properties of an object	256
Get a property	256
Change properties of an object	257
Execute a DTML Method or DTML Document	257
Traverse to an object and add a new property	257
Add a new object to the context	257
DTML versus Python versus Perl versus Page Templates	258
Remote Scripting and Network Services	258
Using XML-RPC	259
Remote Scripting with HTTP	260
Conclusion	261
Zope Services	262
Access Rule Services	262
Temporary Storage Services	263
Version Services	263
Caveat: Versions and ZCatalog	265
Caching Services	265
Adding a Cache Manager	266
Caching an Object	267
Outbound Mail Services	268
Error Logging Services	268
Virtual Hosting Services	269
Searching and Indexing Services	269
Sessioning Services	269
Internationalization Services	269
Searching and Categorizing Content	270
Getting started with Mass Cataloging	270
Creating a ZCatalog	270
Creating Indexes	271
Finding and Cataloging Objects	273
Search and Report Forms	273
Configuring ZCatalogs	274

The Zope Book (2.6 Edition)

Defining Indexes	274
Defining Meta Data	276
Searching ZCatalogs	276
Searching with Forms	277
Searching from Python	278
Searching and Indexing Details	278
Searching ZCTextIndexes	279
Boolean expressions	279
Parentheses	279
Wild cards	279
Phrase search	279
Lexicons	280
Lexicons can:	280
Searching Field Indexes	280
Searching Keyword Indexes	282
Searching Path Indexes	283
Searching DateIndexes	283
Searching DateRangeIndexes	283
Searching TopicIndexes	283
Advanced Searching with Records	284
Keyword Index Record Attributes	284
FieldIndex Record Attributes	284
Allowed values:	285
Path Index Record Attributes	285
DateIndex Record Attributes	286
Allowed values:	287
DateRangeIndex Record Attributes	287
TopicIndex Record Attributes	287
ZCTextIndex Record Attributes	287
Creating Records in HTML	287
Automatic Cataloging	288
Conclusion	294
Relational Database Connectivity	296
Common Relational Databases	296
Database Adapters	297
Setting up a Database Connection	297
Z SQL Methods	300

Examples of ZSQL Methods	300
Displaying Results from Z SQL Methods	303
Providing Arguments to Z SQL Methods	304
Dynamic SQL Queries	305
Inserting Arguments with the Sqlvar Tag	306
Equality Comparisons with the sqltest Tag	306
Creating Complex Queries with the sqlgroup Tag	307
Advanced Techniques	309
Calling Z SQL Methods with Explicit Arguments	309
Acquiring Arguments from other Objects	309
Traversing to Result Objects	310
Other Result Object Methods	311
Binding Classes to Result Objects	312
Caching Results	314
Transactions	315
Further help	316
Summary	316
Virtual Hosting Services	317
Virtual Host Monster	317
Where to Put a Virtual Host Monster And What To Name It	317
Special VHM Path Elements VirtualHostBase and VirtualHostRoot	317
VirtualHostBase	318
VirtualHostRoot	318
Using VirtualHostRoot and VirtualHostBase Together	319
Testing a Virtual Host Monster	319
Arranging for Incoming URLs to be Rewritten	320
Virtual Host Monster Mappings Tab	320
Apache Rewrite Rules	321
"Inside-Out" Virtual Hosting	322
Sessions	323
Introduction	323
Session Configuration	323
Using Session Data	324
Details	325
Terminology	326
Default Configuration	326
Advanced Development Using Sessioning	326

The Zope Book (2.6 Edition)

Overview	326
Obtaining A Session Data Object	327
Modifying A Session Data Object	327
Manually Invalidating A Session Data Object	327
Manually Invalidating A Browser Id Cookie	328
An Example Of Using Session Data from DTML	328
Using the mapping Keyword With A Session Data Object in a dtml-with	328
Using Session Data From Python	329
Interacting with Browser Id Data	329
Determining Which Namespace Holds The Browser Id	330
Obtaining the Browser Id Name/Value Pair and Embedding It Into A Form	330
Determining Whether A Browser Id is "New"	331
Determining Whether A Session Data Object Exists For The Browser Id Associated With This Request	331
Embedding A Browser Id Into An HTML Link	331
Using Session onAdd and onDelete Events	332
Writing onAdd and onDelete Methods	333
Configuration and Operation	334
Setting Initial Transient Object Container Parameters	334
Instantiating Multiple Browser Id Managers (Optional)	334
Instantiating A Session Data Manager (Optional)	336
Instantiating a Transient Object Container	336
Configuring Sessioning Permissions	337
Permissions related to browser id managers:	337
Permissions related to session data managers:	337
Permissions related to transient object containers:	338
Concepts and Caveats	338
Security Considerations	338
Browser Id (Non-)Expiration	338
Session Data Object Expiration Considerations	339
Sessioning and Transactions	339
Mutable Data Stored Within Session Data Objects	339
Session Data Object Keys	340
In-Memory Session Data Container RAM Utilization	340
Mounted Transient Object Container Caveats	340
Conflict Errors	340
Zope Versions and Sessioning	341

The Zope Book (2.6 Edition)

Further Documentation	341
Scalability and ZEO	342
What is ZEO?	342
When you should use ZEO	343
Installing and Running ZEO	344
How to Run Multiple ZEO Clients	345
How to Distribute Load	346
User Chooses a Mirror	346
Using Round-robin DNS to Distribute Load	348
Using Layer 4 Switching to Distribute Load	348
Dealing with the Storage Server as A Single Point of Failure	349
ZEO Server Details	350
ZEO Caveats	351
Conclusion	352
Managing Zope Objects Using External Tools	353
General Caveats	353
FTP and WebDAV	354
Using FTP to Manage Zope Content	354
Determining Your Zope's FTP Port	354
Transferring Files with WS_FTP	355
Remote Editing with FTP/DAV-Aware Editors	355
Editing Zope Objects with Emacs FTP Modes	356
Caveats With FTP	357
Editing Zope Objects with WebDAV	357
Note	357
Using a PUT_factory to Specify the Type of Objects Created With FTP and DAV	358
Using The External Editor Product	359
Other Integration Facilities	360
Chapter 14: Extending Zope	361
Creating Zope Products	361
Creating A Simple Product	362
Creating ZClasses	365
Creating Views of Your ZClass	367
Creating Views of Your ZClass	368
Creating Properties on Your ZClass	369
Creating Methods on your ZClass	371

The Zope Book (2.6 Edition)

ObjectManager ZClasses	373
ZClass Security Controls	373
Controlling access to Methods and Property Sheets	374
Controlling Access to instances of Your ZClass	375
Providing Context-Sensitive Help for your ZClass	375
Using Python Base Classes	376
Distributing Products	377
Maintaining Zope	379
Starting Zope Automatically at Boot Time	379
Debug Mode and Automatic Startup	379
Linux	379
Distributions with Prepackaged Zope	379
Automatic Startup for Custom-Built Zopes	380
This script lets you perform start / stop / restart operations:	384
Mac OS X	384
MS Windows	384
Installing New Products	384
Server Settings	385
Database Cache	385
Interpreter Check Intervals	386
ZServer Threads	386
Database Connections	387
Signals (POSIX only)	387
Monitoring	388
Monitor the Event Log and the Access Log	388
Monitor the HTTP Service	388
Log Files	389
Access Log	389
Event Log	389
Log Rotation	389
Packing and Backing Up the FileStorage Database	390
Database Recovery Tools	391
Appendix A: DTML Reference	393
call: Call a method	393
Syntax	393
Examples	393
See Also	393

comment: Comments DTML	393
Syntax	393
Examples	393
functions: DTML Functions	394
Functions	394
Attributes	397
See Also	397
string module	397
random module	397
math module	397
sequence module	397
Built-in Python Functions	397
if: Tests Conditions	397
Syntax	397
Examples	397
See Also	398
in: Loops over sequences	398
Syntax	398
Attributes	398
Tag Variables	399
Current Item Variables	399
Summary Variables	400
Grouping Variables	400
Batch Variables	400
Examples	401
let: Defines DTML variables	402
Syntax	402
Examples	402
See Also	403
mime: Formats data with MIME	403
Syntax	403
Attributes	403
Examples	404
See Also	404
raise: Raises an exception	404
Syntax	404
Examples	404

See Also	404
return: Returns data	405
Syntax	405
Examples	405
sendmail: Sends email with SMTP	405
Syntax	405
Attributes	405
Examples	406
See Also	406
sqlgroup: Formats complex SQL expressions	406
Syntax	406
Attributes	406
Examples	406
See Also	407
sqltest: Formats SQL condition tests	407
Syntax	408
Attributes	408
Examples	408
See Also	409
sqlvar: Inserts SQL variables	409
Syntax	409
Attributes	409
Examples	409
See Also	409
tree: Inserts a tree widget	409
Syntax	409
Attributes	410
Tag Variables	411
Tag Control Variables	411
Examples	411
try: Handles exceptions	411
Syntax	411
Attributes	412
Tag Variables	412
Examples	412
See Also	413
unless: Tests a condition	413

Syntax	413
Examples	413
See Also	413
var: Inserts a variable	413
Syntax	414
Attributes	414
Examples	415
with: Controls DTML variable look up	415
Syntax	416
Attributes	416
Examples	416
See Also	416
Appendix B: API Reference	417
module AccessControl	417
AccessControl: Security functions and classes	417
class SecurityManager	417
calledByExecutable(self)	417
validate(accessed=None, container=None, name=None, value=None, roles=None)	417
checkPermission(self, permission, object)	417
getUser(self)	417
validateValue(self, value, roles=None)	418
def getSecurityManager()	418
Returns the security manager. See the SecurityManager class.	418
module AuthenticatedUser	418
class AuthenticatedUser	418
getUserName()	418
getId()	418
has_role(roles, object=None)	418
getRoles()	418
has_permission(permission, object)	418
getRolesInContext(object)	419
getDomains()	419
module DTMLDocument	419
class DTMLDocument(ObjectManagerItem, PropertyManager)	419
manage_edit(data, title)	419
document_src()	419
__call__(client=None, REQUEST={}, RESPONSE=None, **kw)	419

From DTML	420
From Python	420
By the Publisher	420
get_size()	420
ObjectManager Constructor	420
manage_addDocument(id, title)	420
module DTMLMethod	421
class DTMLMethod(ObjectManagerItem)	421
manage_edit(data, title)	421
document_src()	421
__call__(client=None, REQUEST={}, **kw)	421
From DTML	422
From Python	422
By the Publisher	422
get_size()	422
ObjectManager Constructor	422
manage_addDTMLMethod(id, title)	422
module DateTime	422
class DateTime	422
strftime(format)	425
Return date time string formatted according to format	425
dow()	425
aCommon()	425
h_12()	425
Mon_()	425
HTML4()	425
greaterThanEqualTo(t)	425
dayOfYear()	426
lessThan(t)	426
AMPM()	426
isCurrentHour()	426
Month()	426
mm()	426
ampm()	426
hour()	427
aCommonZ()	427
Day_()	427

The Zope Book (2.6 Edition)

pCommon()	427
minute()	427
day()	427
earliestTime()	427
Date()	427
Time()	427
isFuture()	428
greaterThan(t)	428
TimeMinutes()	428
yy()	428
isCurrentDay()	428
dd()	428
rfc822()	428
isLeapYear()	429
fCommon()	429
isPast()	429
fCommonZ()	429
timeTime()	429
toZone(z)	429
lessThanEqualTo(t)	429
Mon()	429
parts()	430
isCurrentYear()	430
PreciseAMPM()	430
AMPMMinutes()	430
equalTo(t)	430
pDay()	430
notEqualTo(t)	430
h_24()	430
pCommonZ()	431
isCurrentMonth()	431
DayOfWeek()	431
latestTime()	431
dow_1()	431
timezone()	431
year()	431
PreciseTime()	431

ISO()	432
millis()	432
second()	432
month()	432
pMonth()	432
aMonth()	432
isCurrentMinute()	432
Day()	432
aDay()	433
module ExternalMethod	433
class ExternalMethod	433
manage_edit(title, module, function, REQUEST=None)	433
__call__(*args, **kw)	433
ObjectManager Constructor	433
manage_addExternalMethod(id, title, module, function)	433
module File	434
class File(ObjectManagerItem, PropertyManager)	434
getContentType()	434
update_data(data, content_type=None, size=None)	434
getSize()	435
ObjectManager Constructor	435
manage_addFile(id, file="", title="", precondition="", content_type="")	435
Creates a new File object id with the contents of file	435
module Folder	435
class Folder(ObjectManagerItem, ObjectManager, PropertyManager)	435
ObjectManager Constructor	435
manage_addFolder(id, title)	435
module Image	435
class Image(File)	435
tag(height=None, width=None, alt=None, scale=0, xscale=0, yscale=0, **args)	436
ObjectManager Constructor	436
manage_addImage(id, file, title="", precondition="", content_type="")	436
module MailHost	436
class MailHost	436
send(messageText, mto=None, mfrom=None, subject=None, encode=None)	436
simple_send(self, mto, mfrom, subject, body)	437
MailHost Constructor	437

manage_addMailHost(id, title="", smtp_host=None, localhost=localhost, smtp_port=25, timeout=1.0)	437
module ObjectManager	437
class ObjectManager	437
objectItems(type=None)	438
superValues(type)	438
objectValues(type=None)	438
objectIds(type=None)	439
module ObjectManagerItem	439
class ObjectManagerItem	439
title_or_id()	439
getPhysicalRoot()	439
manage_workspace()	440
getPhysicalPath()	440
unrestrictedTraverse(path, default=None)	440
getId()	440
absolute_url(relative=None)	440
this()	440
restrictedTraverse(path, default=None)	441
title_and_id()	441
module PropertyManager	441
class PropertyManager	441
propertyItems()	441
propertyValues()	441
propertyMap()	441
propertyIds()	442
getPropertyType(id)	442
getProperty(id, d=None)	442
hasProperty(id)	442
module PropertySheet	442
class PropertySheet	442
xml_namespace()	442
propertyItems()	442
propertyValues()	442
getPropertyType(id)	443
propertyInfo()	443
getProperty(id, d=None)	443

manage_delProperties(ids=None, REQUEST=None)	443
manage_changeProperties(REQUEST=None, **kw)	443
manage_addProperty(id, value, type, REQUEST=None)	444
propertyMap()	444
propertyIds()	444
hasProperty(id)	445
module PropertySheets	445
class PropertySheets	445
get(name, default=None)	445
values()	445
items()	445
module PythonScript	445
class PythonScript(Script)	445
document_src(REQUEST=None, RESPONSE=None)	447
ZPythonScript_edit(params, body)	447
ZPythonScript_setTitle(title)	448
ZPythonScriptHTML_upload(REQUEST, file="")	448
write(text)	448
ZScriptHTML_tryParams()	448
read()	448
ZPythonScriptHTML_editAction(REQUEST, title, params, body)	448
ObjectManager Constructor	448
manage_addPythonScript(id, REQUEST=None)	448
module Request	448
class Request	449
get_header(name, default=None)	450
items()	450
keys()	450
setVirtualRoot(path, hard=0)	450
values()	451
set(name, value)	451
has_key(key)	451
setServerURL(protocol=None, hostname=None, port=None)	451
module Response	451
class Response	451
setHeader(name, value)	451
setCookie(name, value, **kw)	451

addHeader(name, value)	452
appendHeader(name, value, delimiter=,)	452
write(data)	452
setStatus(status, reason=None)	452
setBase(base)	452
expireCookie(name, **kw)	452
appendCookie(name, value)	453
redirect(location, lock=0)	453
class Script	453
ZScriptHTML_tryAction(REQUEST, argvars)	453
module SessionInterfaces	453
Session API	453
class SessionDataManagerErr	453
class BrowserIdManagerInterface	454
getBrowserId(self, create=1)	454
isBrowserIdFromCookie(self)	454
isBrowserIdNew(self)	454
encodeUrl(self, url)	454
flushBrowserIdCookie(self)	454
getBrowserIdName(self)	455
isBrowserIdFromForm(self)	455
hasBrowserId(self)	455
setBrowserIdCookieByForce(self, bid)	455
class BrowserIdManagerErr	455
class SessionDataManagerInterface	455
getSessionDataByKey(self, key)	456
getSessionData(self, create=1)	456
getBrowserIdManager(self)	456
hasSessionData(self)	456
module TransienceInterfaces	456
class TransientObject	456
delete(self, k)	457
setLastAccessed(self)	457
getCreated(self)	457
values(self)	457
has_key(self, k)	457
getLastAccessed(self)	457

getId(self)	457
update(self, d)	457
clear(self)	458
items(self)	458
keys(self)	458
get(self, k, default=marker)	458
set(self, k, v)	458
getContainerKey(self)	458
invalidate(self)	458
class MaxTransientObjectsExceeded	458
class TransientObjectContainer	459
new(self, k)	459
setDelNotificationTarget(self, f)	459
getTimeoutMinutes(self)	459
has_key(self, k)	459
setAddNotificationTarget(self, f)	459
getId(self)	460
setTimeoutMinutes(self, timeout_mins)	460
new_or_existing(self, k)	460
get(self, k, default=None)	460
getAddNotificationTarget(self)	460
getDelNotificationTarget(self)	460
module UserFolder	460
class UserFolder	461
userFolderEditUser(name, password, roles, domains, **kw)	461
userFolderDelUsers(names)	461
userFolderAddUser(name, password, roles, domains, **kw)	461
getUsers()	461
getUserNames()	461
getUser(name)	461
module Vocabulary	461
class Vocabulary	461
words()	462
insert(word)	462
query(pattern)	462
ObjectManager Constructor	462
manage_addVocabulary(id, title, globbing=None, REQUEST=None)	462

module ZCatalog	462
class ZCatalog	462
schema()	462
__call__(REQUEST=None, **kw)	463
uncatalog_object(uid)	463
getobject(rid, REQUEST=None)	463
indexes()	463
getpath(rid)	463
index_objects()	463
searchResults(REQUEST=None, **kw)	463
There are some rules to consider when querying this method:	464
uniqueValuesFor(name)	464
catalog_object(obj, uid)	464
ObjectManager Constructor	464
manage_addZCatalog(id, title, vocab_id=None)	464
module ZSQLMethod	464
class ZSQLMethod	464
manage_edit(title, connection_id, arguments, template)	465
__call__(REQUEST=None, **kw)	465
ObjectManager Constructor	465
manage_addZSQLMethod(id, title, connection_id, arguments, template)	465
module ZTUtils	466
ZTUtils: Page Template Utilities	466
class Batch	466
__init__(self, sequence, size, start=0, end=0, orphan=0, overlap=0)	466
module math	467
math: Python math module	467
See Also	467
module random	467
random: Python random module	467
See Also	467
module sequence	467
sequence: Sequence sorting module	467
def sort(seq, sort)	467
DTML Examples	468
Page Template Examples	468
See Also	468

module standard	468
Products.PythonScripts.standard: Utility functions and classes	468
def structured_text(s)	469
See Also	469
def html_quote(s)	469
See Also	469
def url_quote_plus(s)	469
See Also	469
def dollars_and_cents(number)	469
def sql_quote(s)	469
def whole_dollars(number)	469
def url_quote(s)	469
See Also	469
class DTML	470
__init__(source, **kw)	470
call(client=None, REQUEST={}, **kw)	470
def thousand_commas(number)	470
def newline_to_br(s)	470
module string	470
string: Python string module	470
See Also	470
Appendix C: Zope Page Templates Reference	472
TAL Overview	472
TAL Namespace	472
TAL Statements	472
Order of Operations	473
See Also	473
attributes: Replace element attributes	474
Syntax	474
Description	474
Examples	474
condition: Conditionally insert or remove an element	474
Syntax	475
Description	475
Examples	475
content: Replace the content of an element	475
Syntax	475

Description	475
Examples	475
See Also	476
define: Define variables	476
Syntax	476
Description	476
Examples	476
omit-tag: Remove an element, leaving its contents	476
Syntax	476
Description	477
Examples	477
on-error: Handle errors	477
Syntax	477
Description	477
Examples	478
See Also	478
repeat: Repeat an element	478
Syntax	478
Description	478
Repeat Variables	479
The following information is available from the repeat variable:	479
Examples	480
replace: Replace an element	480
Syntax	480
Description	480
Examples	481
See Also	481
TALES Overview	481
TALES Expression Types	481
Built-in Names	482
See Also	482
TALES Exists expressions	483
Syntax	483
Description	483
Examples	483
TALES Nocall expressions	483
Syntax	483

The Zope Book (2.6 Edition)

Description	483
Examples	484
TALES Not expressions	484
Syntax	484
Description	484
Examples	484
TALES Path expressions	484
Syntax	485
Description	485
Examples	485
TALES Python expressions	486
Syntax	486
Description	486
Security Restrictions	486
Built-in Functions	486
Python Modules	487
Examples	487
TALES String expressions	487
Syntax	488
Description	488
Examples	488
METAL Overview	488
METAL Namespace	488
METAL Statements	488
See Also	489
define-macro: Define a macro	489
Syntax	489
Description	489
Examples	489
See Also	490
define-slot: Define a macro customization point	490
Syntax	490
Description	490
Examples	490
See Also	490
fill-slot: Customize a macro	490
Syntax	490

The Zope Book (2.6 Edition)

Description	490
Examples	491
See Also	491
use-macro: Use a macro	491
Syntax	491
Description	491
Examples	491
See Also	492
ZPT-specific Behaviors	492
HTML Support Features	492
Appendix D: Zope Resources	494
Zope Web Sites	494
Zope Documentation	494
(Other) Zope Books	494
Mailing Lists	494
Python Information	494
DTML Name Lookup Rules	495

Preface

Welcome to *The Zope Book* . This book is designed to introduce you to Zope, the open source web application server.

To make effective use of the book, you should know how to use a web browser and you should have a basic understanding of HTML (Hyper Text Markup Language) and URLs (Uniform Resource Locators). You don't need to be a highly-skilled programmer in order to use Zope, but some programming background (particularly object-oriented programming) will be extremely helpful.

How the Book Is Organized

A brief summary of each chapter is presented below.

1. Introducing Zope

This chapter explains what Zope is and what it can do for you. You also learn about the differences between Zope and other web application servers.

2. Zope Concepts and Architecture

This chapter explains fundamental Zope concepts and describes some of Zope's architecture.

3. Installing and Starting Zope

This chapter explains how to install and start Zope for the first time. By the end of this chapter, you should have Zope installed and working.

4. Object Orientation

This chapter explains the concept of *object orientation* , which is the development methodology most often used to create Zope applications.

5. Using The Zope Management Interface

This chapter explains how to use Zope's web-based management interface. By the end of this chapter you should be able to navigate around the Zope object space, copy and move objects, and use other basic Zope features.

6. Using Basic Zope Objects

This chapter introduces *objects* , which are the most important elements of Zope. We introduce the basic Zope objects: content objects, presentation objects, and logic objects, and we build a simple application using these objects.

7. Acquisition

This chapter introduces *acquisition* , which is Zope's mechanism for sharing site behavior and content via "containment".

8. Basic DTML

This chapter introduces *DTML* , Zope's tag-based scripting language. We describe how to use DTML's templating and scripting facilities. 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.

9. Using Zope Page Templates

This chapter introduces Zope Page Templates, another Zope tool used to create dynamic web pages. This chapter shows you how to create and edit page templates. It also introduces basic template statements that let you insert dynamic content.

10. Creating Basic Zope Applications

This chapter walks the reader through several real-world examples of building a Zope application. It explains how to use basic Zope objects and how they can work together to form basic applications.

11. Users and Security

This chapter looks at how Zope handles users, authentication, authorization, and other security-related matters.

12. Advanced DTML

This chapter 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.

13. Advanced Page Templates

This chapter 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.

14. Advanced Zope Scripting

This chapter covers scripting Zope with Python and Perl. It explains how to write business logic in Zope using tools more powerful 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.

15. Zope Services

This chapter covers Zope objects which are "services" that don't readily fit into any of the basic "content", "presentation" or "logic" object groups.

16. Searching and Categorizing Content

This chapter 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 metadata and search results.

17. Relational Database Connectivity

This chapter describes how Zope connects to external relational 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.

18. Virtual Hosting Services

This chapter explains how to set up Zope in a "virtual-hosted" environment where Zope subfolders can be served as "top-level" hostnames. It includes examples that allow virtual hosting to be performed "natively" or using Apache's `mod_rewrite` facility.

19. Sessions

This chapter describes Zope's "sessioning" services, which allow Zope developers to "keep state" between HTTP requests.

20. Scalability and ZEO

This chapter 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 many simultaneous visitors.

21. Managing Zope Objects Using External Tools

This chapter explains how to use tools other than your web browser to manipulate Zope objects.

22. Extending Zope

This chapter covers extending Zope by creating your own classes of objects. It discusses *ZClasses*, and how instances are built from classes. It describes how to build a ZClass and its attendant security and design issues. Finally, it discusses creating Python base classes for ZClasses and describes the base classes that ship with Zope.

23. Maintaining Zope

This chapter covers Zope maintenance and administration tasks such as database "packing" and Product installation.

24. Appendix A: DTML Reference

Reference of DTML syntax and commands.

25. Appendix B: API Reference

Reference of Zope object APIs.

26. Appendix C: Page Template Reference

Reference of Zope Page Template syntax and commands.

27. Appendix D: Zope Resources

Reference of "resources" which can be used to further enhance your Zope learning experience.

28. Appendix E: DTML Name Lookup Rules

Describes DTML's name lookup rules.

Conventions Used in This Book

This book uses the following typographical conventions:

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

Fixed width — Fixed width text indicates objects, commands, hyperlinks, and code listings.

Contributors to This Book

Contributors to this book include Amos Latteier, Michel Pelletier, Chris McDonough, Evan Simpson, Tom Deprez, Paul Everitt, Bakhtiar A. Hamid, Geir Baekholt, Paul Winkler, Peter Sabaini, Andrew Veitch, Kevin Carlson and the Zope Community.

Amos and Michel wrote the entirety of the first edition of this book, and kept the online version of the book current up until Zope 2.5.1.

Tom Deprez provided much-needed editing assistance on the first book edition.

Evan Simpson edited the chapters related to ZPT for the 2.6 edition.

Paul Everitt contributed to the first few chapters of the first edition, edited the first few chapters of the second edition for sanity and contributed some "Maintaining Zope" content for the 2.6 edition.

Bakhtiar Hamid edited the ZEO chapter for the 2.6 edition.

Geir edited and extended the Users and Security chapter for the 2.6 edition.

Paul Winkler with help from Peter Sabaini expertly massaged the Advanced Scripting chapter into coherency for the 2.6 edition.

Peter Sabaini greatly fleshed out and extended the "Maintaining Zope" chapter for the 2.6 Edition.

Andrew Veitch cheerfully performed the thankless task of editing and extending the Relational Database Connectivity chapter for the 2.6 edition.

Kevin Carlson masterfully edited and expanded the Advanced DTML chapter.

Chris McDonough edited the entirety of the book for the 2.6 edition, entirely rewrote a few chapters and added new material related to object orientation, using the Zope management interface, acquisition, installation, services, virtual hosting, sessions, and DTML name lookup rules.

Anyone who added a comment to the online BackTalk edition of the first online edition of this book contributed greatly. Thank you!

Introducing Zope

Zope is a framework that allows developers of varying skill levels to build *web applications*. This chapter explains Zope's purpose and audience in greater detail. It also describes what makes Zope different from similar applications.

What Is A Web Application?

It is often important that visitors to a website see content that is timely and up-to-date. A time-dependent site's content needs to change continually. For example, if a commercial website helps its visitors sell and buy used automobiles, it is usually required that the site run advertisements only for cars that have not yet been sold. It is also important that new ads be posted at most a day or two after they've been placed by a seller. If either of these requirements is not met, the website will likely not be very successful.

The layout of text and images that show up in a user's web browser when the user visits a website are often composed using a simple language known as Hyper Text Markup Language (HTML). When a user visits a typical website, a chunk of text that is "marked-up" with HTML is transferred between the website and the user's browser. The browser interprets the chunk of text, showing text and images to the user. The chunk of text which is transferred is typically referred to as a *page*. Many website visitors think about navigation in terms of moving "from page-to-page" within a website. When they click on a hyperlink, their browser transports them to a new page. When they hit their browser's *Back* button, it takes them to the last page they've visited.

Some websites are *static*. Static websites require a person with a privileged level of access (sometimes termed the *webmaster*) to manually "freshen" the site's content. Freshening the content requires the person to manually visit and update the HTML that makes up each page that needs to change. Typically, this is done by editing a set of files on the *web server* (the machine that runs the website), where each file represents a single page.

Site-wide changes to the "look-and-feel" of a static website require that the webmaster visit and update each and every file that comprises the website. Websites can typically grow to encompass thousands of files, so this can become a non-trivial task. The webmaster responsible for our automobile advertising website has the additional responsibility of keeping the ads themselves fresh. If each page in the website represents an ad for a particular automobile, he needs to delete the pages representing ads which have expired and create new pages for ads which have been recently sold. He then needs to make sure that no hyperlinks on other pages point to missing pages.

This becomes a lot of work very quickly. As you can imagine, with any more than a few pages to update every day, this can become pretty dull. The webmaster also understandably makes mistakes (he's human, after all), and forgets to update or remove critical pages.

Somewhere down the line smart webmasters begin to think to themselves, "Wow, this is a lot of work. It's tedious and complicated, and I seem to be making a lot of mistakes. Computers are really good at doing tedious and complicated tasks, and they don't make very many mistakes. I bet my webserver computer can automatically do a lot of the work I now do manually." At this point, the webmaster is ready to be introduced to *web applications*.

A *web application* is a computer program that users invoke by using a web browser to contact a web server via the Internet. Users and browsers are typically unaware of the difference between contacting a web server which fronts for a statically-built website and a web server which fronts for a web application. But unlike a static website, a web application creates its "pages" *dynamically*. A website that is dynamically-constructed uses an a computer program to provide the dynamism. These kinds of dynamic applications can be written in any number of computer languages.

In a dynamically-constructed website, the webmaster is not required to visit the site "page-by-page" in order to update content or style. Instead, he is able to create a "common look and feel" for the set of pages that make up his website. He is also able to instruct the webserver to generate an HTML page on the fly that includes certain unique bits of

content. If our auto-classified-ad webmaster chose to construct a web application to maintain his classifieds system, he could maintain a list of "current" ads separate from the HTML layout (perhaps stored in a database of some kind). He could then instruct his web application to query this database and generate a particular chunk of HTML that represented an ad or an index of ads when a user visited a page in his website.

Web applications are everywhere. Common examples of web applications are those that let you search the web, like *Google* ; collaborate on projects, like *SourceForge* ; buy items at an auction like *eBay* ; communicate with other people over e-mail, like *Hotmail* ; or view the latest news ala *CNN.com* .

A framework which allows people to construct a web application is often called a *web application server* , or sometimes just an *application server* . Zope is a web application server, as are competing products like BEA WebLogic , Macromedia ColdFusion and (to some extent) Vignette StoryServer . A web application server typically allows a developer to create a web application using some common computer programming language. But it also provides services beyond the basic capabilities of the language such as templating, a common security model, data persistence, sessions, and other features that people find useful when constructing a typical web application.

How You Can Benefit From Using An Application Server

If you are considering writing even a moderately-sized web application, it is typically a good idea to start your project using an application server framework unless your application requirements are extremely specialized. By starting a web application project using an application server framework (as opposed to a "raw" computer language such as Perl, Python, TCL, or C), you are able to utilize the services of the framework that have already been written, and you avoid needing to write the functionality yourself "from scratch" in a "raw" language.

Many application servers allow you perform some of the below-mentioned tasks.

Present Dynamic Content — You may tailor your web site's presentation to its users and provide users with search features. Application servers allow you to serve dynamic content. Application servers typically come with facilities for personalization, database integration and content indexing and search.

Manage Your Web Site — A small web site is easy to manage, but a web site that serves thousands of documents, images and files requires heavy-duty management tools. It is useful to be able to manage your site's data, business logic and presentation from a single place. An application server can typically help manage your content and presentation in this way.

Build a Content Management System — A fairly new breed of application, a *content management system* allows nontechnical editors to create and manage content for your website. Application servers provide the tools with which you can build a content management system.

Build an E-Commerce Application — Application servers provide a framework in which sophisticated e-commerce applications can be created.

Securely Manage Contributor Responsibility — When you deal with more than a handful of web users, security becomes very important. It is important to be able to safely delegate tasks to different classes of system users. 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. Application servers typically provide a mechanism for access control and delegation.

Provide Network Services — You may want to produce or consume *network services* . A network service-enabled web site will need to be able to accept requests from other computer programs. For example, if your website is a news site, you may wish to share your news stories with another website; you can do this by making the news feed a network service. Or perhaps you want to make products for sale on your site automatically searchable from a product comparison site. Application servers are beginning to offer methods of enabling these kinds of network services.

Integrate Diverse Systems — Your existing content may be contained in many places: relational databases, files, separate web sites, and so on. Application servers typically allow you to present a unified view of your existing data by integrating diverse third-party systems.

Provide Scalability — Application servers allow your web applications to scale across as many systems as necessary to handle the load demands of your websites.

The Zope application server allows you to perform all of these tasks.

Zope History

In 1996 Jim Fulton (the current CTO of Zope Corporation, the distributors of Zope) was drafted to teach a class on CGI programming, despite not knowing very much about the subject. CGI or *common gateway interface* programming is a commonly-used web development model that allows developers to construct dynamic websites. 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. From these initial musings, the core of Zope was written on the plane flight back from the class.

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

The moniker "Zope" stands for the *Z Object Publishing Environment* (the "Z" doesn't really mean anything in particular). Most of Zope is written in the Python scripting language, with performance-critical pieces written in C.

Why Use Zope Instead of Another Application Server

If you're in the business of creating web applications, Zope can potentially help you create them at less cost and at a faster rate than you could by using another competing web application server. This claim is backed up by a number of Zope features:

- Zope is free of cost and is distributed under an open-source license. There are many non-free commercial application servers that are relatively expensive.
- Zope itself is an inclusive platform. It ships with all the necessary components to begin developing an application. You don't need to license extra software to support Zope (e.g. a relational database) in order to develop your application. This also makes Zope very easy to install. Many other application servers have "hidden" costs by requiring that you license expensive software or to configure complex third-party infrastructure software before you can begin to develop your application.
- Zope allows and encourages third-party developers to package and distribute ready-made applications. Due to this, Zope has a wide variety of integrated services and add-on products available for immediate use. Most of these components, like Zope itself, are free and open source. Zope's popularity has bred a large community of application developers. Many other application servers do not have a large base of third-party support or a means for so neatly packaging plug-ins.
- Applications created in Zope can scale almost linearly using Zope's Zope Enterprise Objects (ZEO) clustering solution. Using ZEO, you can deploy a Zope application across many physical computers without needing to

change much (if any) of your application code. Many application servers don't scale quite as transparently or as predictably.

- Zope allows developers to create web applications using only a web browser. The Internet Explorer, Mozilla, Netscape, OmniWeb, Konqueror, and Opera browsers are all known to be able to be used to display and manipulate Zope's development environment (the *Zope Management Interface* also known as the *ZMI*). Zope also allows developers to safely delegate application development duties to other developers "through the web" using the same interface. Very few other application servers, if any, deliver the same level of functionality.
- Zope provides a granular and extensible security framework. You can easily integrate Zope with diverse authentication and authorization systems such as LDAP, Windows NT, and RADIUS simultaneously, using prebuilt modules. Many other application servers lack support for some important authentication and authorization systems.
- Zope allows teams of developers to collaborate effectively. Collaborative environments require tools to allow users to work without interfering with each other, so Zope has *Undo*, *Versions*, *History* and other tools to help people work safely together and recover from mistakes. Many other application servers do not provide these kinds of features.
- Zope runs on most popular microcomputer operating system platforms: Linux, Windows NT/2000/XP, Solaris, FreeBSD, NetBSD, OpenBSD, and Mac OS X. Zope even runs on Windows 98/ME (recommended only for development purposes, however). Many other application server platforms require that you run an operating system of their licensor's choosing.
- Zope can be extended using the interpreted Python scripting language. Python is popular and easy to learn, and it promotes rapid development. Many libraries are available for Python which can be used when creating your own application. Many other application servers must be extended using compiled languages such as Java, which cuts down on development speed. Many other application servers use less popular languages for which there are not as many ready-to-use library features.

For examples of applications that have already been created using Zope, please see Zope Corporation's case studies page online at Zope.com.

Zope Audiences and What Zope Isn't

Managing the development process of a large-scale site can be a difficult task. It often takes many people working together to create, deploy, and manage a web application.

- *Information Architects* make platform decisions and keep track of the "big picture".
- *Component Developers* create software intended for reuse and distribution.
- *Site Developers* integrate the software written by component developers and native application server services, building an application in the process.
- *Site Designers* create the site's look and feel.
- *Content Managers* create and manage the site's content.
- *Administrators* keep the software and environment running.

- *Consumers* use the site to locate and work with useful content.

Of the parties listed above, Zope is most useful for *component developers* , *site developers* , and *site designers* . These three groups of people can collaborate to produce an application using Zope's native services and third-party *Zope Products* . They will typically produce applications useful to *content managers* and *consumers* under the guide of the *information architect* . *Administrators* will deploy the application and tend to the application after it is has been created.

Note that Zope is a web application construction framework that programmers of varying skill levels may use to create web-based applications. It *is not* itself an application that is ready to use "out of the box" for any given application. For example, Zope itself is not a weblog, a content management system, or a "e-shop-in-a-box" application.

However, freely available *Products* built on top of Zope offer these kinds of services. At the time of this writing, the Zope.org website catalogs roughly 500 Products that you can browse and hopefully reuse in your own application. There are Products for weblogging, content management, and ecommerce among these.

Zope is not a visual design tool. Tools like Macromedia Dreamweaver or Adobe GoLive allow designers to create "look and feel". You may use these tools to manage Zope-based web sites, but Zope itself does not replace them. You can edit content "through the web" using Zope but the limitations of current cross-platform browser technology prevents Zope from doing as good of a job as these kinds of tools to design web presentation.

Zope's Terms of Use and License and an Introduction to The Zope Community

Zope is free of cost. You are permitted to use Zope to create and run your web applications without paying licensing or usage fees. You may also include Zope in your own products and applications without paying royalty fees to Zope's licensor, *Zope Corporation* .

Zope is distributed under an open source license, the Zope Public License or ZPL . The terms of the ZPL license stipulate that you will be able to obtain and modify the source code for Zope.

The ZPL is different than another popular open source license, the GNU Public License . The licensing terms of the GPL require that if you intend to redistribute a GPL-licensed application, and you modify or extend the application in a meaningful way, that you contribute your modifications back to the licensor. This is not required for ZPL-licensed applications, however. You may modify and retribute Zope without contributing your modifications back to Zope Corporation as long as you follow the other terms of the license faithfully.

Note that the ZPL has been certified as OSD compliant by the Open Source Initiative and is listed as GPL compliant by the Free Software Foundation .

A community of developers is responsible for maintaining and extending the Zope application server. Many community members are professional consultants, developers and web masters who develop applications using Zope for their own gain. Others are students and curious amateur site developers. Zope Corporation is a member of this community. Zope Corporation controls the distribution of the defacto "canonical" Zope version and permits its developers as well as other selected developers to modify this distribution's source code.

The Zope 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 Zope-related mailing lists at Zope.org's mailing list page .

Zope Corporation makes revenues by using Zope to create web applications for its paying customers, by training prospective Zope developers, by selling support contracts to companies who use Zope, and by hosting Zope-powered websites; it does not make any direct revenues from the distribution of the Zope application server itself.

Zope Concepts and Architecture

Fundamental Zope Concepts

The Zope framework has several fundamental underlying concepts, each of which you should understand to make the most of your Zope experience.

Zope Is A Framework

Zope relieves the developer of most of the onerous details of Web application development such as data persistence, data integrity and access control, allowing you to focus on the problem at hand. It allows you to utilize the services it provides to build web applications more quickly than other languages or frameworks. Zope allows you to write web application logic in the Python language, and provides add-on support for Perl. Zope also comes with two solutions that allow you to "template" text, XML, and HTML: Document Template Markup Language (DTML), and Zope Page Templates (ZPT).

Object Orientation

Unlike common file-based Web templating systems such as ASP or PHP, Zope is a highly "object-oriented" Web development platform. Object orientation is a concept that is shared between many different programming languages, including the Python language in which Zope is implemented. The concept of object orientation may take a little "getting-used-to" if you're an old hand at primarily procedural languages typically used for web scripting such as Perl or PHP, but you should be able to get a grasp on the concepts by reading the Object Orientation chapter and by "learning-by-doing" with respect to the examples in the book.

Object Publishing

The technology that would become Zope was founded on the realization that the Web is fundamentally object-oriented. A URL to a Web resource is really just a path to an object in a set of containers, and the HTTP protocol provides a way to send messages to that object and receive its response.

Zope's object structure is hierarchical, which means that a typical Zope site is composed of objects which contain other objects (which may contain other objects, ad infinitum). URLs map naturally to objects in the hierarchical Zope environment based on their names. For example, the URL `"/Marketing/index.html"` could be used to access the Document object named `"index.html"` located in the Folder object named `"Marketing"`.

Zope's seminal duty is to "publish" the objects you create. The way it does this is conceptually straightforward.

1. Your web browser sends a request to the Zope server. The request specifies a URL in the form `protocol://host:port/path?querystring` , e.g. `http://www.zope.org:8080/Resources?batch_start=100` .
2. Zope separates the URL into its component "host", "port" "path" and "query string" portions (`http://www.zope.org` , `8080` , `/Resources` and `?batch_start=100` , respectively).
3. Zope locates the object in its object database corresponding to the "path" (`/Resources`).
4. Zope "executes" the object using the "query string" as a source of parameters that can modify the behavior of the object. This means that the object may behave differently depending on the values passed in the query string.

5. If the act of executing the object returns a value, the value is sent back to your browser. Typically a given Zope object returns HTML, file data, or image data.

6. The data is interpreted by the browser and shown to you.

Mapping URLs to objects isn't a new idea. Web servers like Apache and Microsoft's IIS do the same thing. They translate URLs to files and directories on a filesystem. Zope similarly maps URLs on to objects in its object database.

A Zope object's URL is based on its "path". It is composed of the `ids` of its containing Folders and the object's `id`, separated by slash characters. For example, if you have a Zope "Folder" object in the root folder called *Bob*, then its path would be `/Bob`. If *Bob* is in a sub-folder called *Uncles* then its URL would be `/Uncles/Bob`.

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

```
/Uncles/Rick  
/Uncles/Danny  
/Uncles/Louis
```

The URL of an object is most simply composed of its `host`, `port`, and `path`. So for the Zope object with the path `/Bob` on the Zope server at `http://localhost:8080`, the URL would be `http://localhost:8080/Bob`. Visiting a URL of a Zope object directly is termed *calling the object through the web*. This causes the object to be evaluated and the result of the evaluation is returned to your web browser.

For a more detailed explanation of how Zope performs object publishing, see the Object Publishing chapter of the *Zope Developer's Guide*.

Through-The-Web Management

To create and work with Zope objects, you use your Web browser to access the Zope management interface. All management and application development can be done completely through the Web using only a browser. The Zope management interface provides a familiar Windows Explorer-like view of the Zope object system. Through the management interface a developer can create and script Zope objects or even define new kinds of objects, without requiring access to the file system of the web server.

Objects can be dropped in anywhere in the object hierarchy. Site managers can work with their objects by clicking on tabs that represent different "views" of an object. These views vary depending on the type of object. A "DTML Method" Zope object, for example, has an "Edit" tab which allows you to edit the document's source, while a "Database Connection" Zope object provides views that let you modify the connection string or caching parameters for the object. All objects also have a "Security" view that allows you to manage access control settings for that object.

Security and Safe Delegation

One of the things that sets Zope apart from other application servers is that it was designed from the start to be tightly coupled not only with the Web object model, but also the Web development model. Today's successful Web applications require the participation of many people across an organization who have different areas of expertise. Zope is specifically designed to accommodate this model, allowing site managers to safely delegate control to design experts, database experts and content managers.

A successful Web site requires the collaboration of many people in an organization: application developers, SQL experts, content managers and often even the end users of the application. On a conventional Web site, maintenance and security can quickly become problematic. How much control do you give to the content manager? How does giving the content manager a login affect your security? What about that SQL code embedded in the ASP files he'll be working on - code that probably exposes your database login?

Objects in Zope provide a much richer set of possible permissions than a conventional file-based system. Permissions vary by object type based on the capabilities of that object. This makes it possible to implement fine-grained access control. For example, you can set access control so that content managers can use "SQL Method" objects, but not change them or even view their source. You can also set restrictions so that a user can only create certain kinds of objects, for instance "Folders" and "DTML Documents" but not "SQL Methods" or other objects.

Zope provides the capability to manage users through the web via "User Folders", which are special folders that contain user information. Several Zope add-ons are available that provide extended types of User Folders that get their user data from external sources such as relational databases or LDAP directories. The ability to add new User Folders can be delegated to users within a subfolder, essentially allowing you to delegate the creation and user management of subsections of your website to semi-trusted users without worrying about those users changing the objects "above" their folder.

Native Object Persistence and Transactions

Zope objects are stored in a high-performance transactional object database known as the Zope Object Database (ZODB). Each Web request is treated as a separate transaction by the object database. If an error occurs in your application during a request, any changes made during the request will be automatically rolled back. The object database also provides multi-level undo, allowing a site manager to "undo" changes to the site with the click of a button. The Zope framework makes all of the details of persistence and transactions totally transparent to the application developer. Relational databases which are used with Zope can also play in Zope's transaction framework.

Acquisition

One of the most powerful aspects of Zope is "Acquisition", and the core concept is simply that:

- Zope objects are contained inside other objects (such as Folders).
- Objects can "acquire" attributes and behavior from their containers.

The concept of acquisition works with all Zope objects, and provides an extremely powerful way to centralize common resources. A commonly used SQL query or snippet of HTML, for example, can be defined in one Folder and objects in subfolders can use it automatically through acquisition. If the query needs to be changed, you can change it in one place without worrying about all of the subobjects that use the query.

Because objects are acquired by starting at the current level in the containment hierarchy and searching upward, it is easy to specialize areas of your site with a minimum of work. If, for example, you had a Folder named "Sports" on your site containing sports-related content, you could create a new header and footer document in the Sports Folder that use a sports-related theme. Content in the Sports folder and its subfolders will then use the specialized sports header and footer found in the "Sports" folder rather than the header and footer from the top-level folder on the site.

Acquisition is explained further in the chapter entitled Acquisition .

Zope Is Extensible

Zope is highly extensible, and advanced users can create new kinds of Zope objects, either by writing new Zope add-ons in Python or by building them completely through the Web. The Zope software provides a number of useful built-in components to help extension authors, including a robust set of framework classes that take care of most of the details of implementing new Zope objects.

A number of Zope add-on products are available that provide features like drop-in Web discussion topics, desktop data publishing, XML tools and e-commerce integration. Many of these products have been written by the highly active members of the Zope community, and most are also open source.

Fundamental Zope Components

Zope consists of several different components that work together to help you build web applications. Zope's fundamental components are shown in the figure below, and explained following the figure.

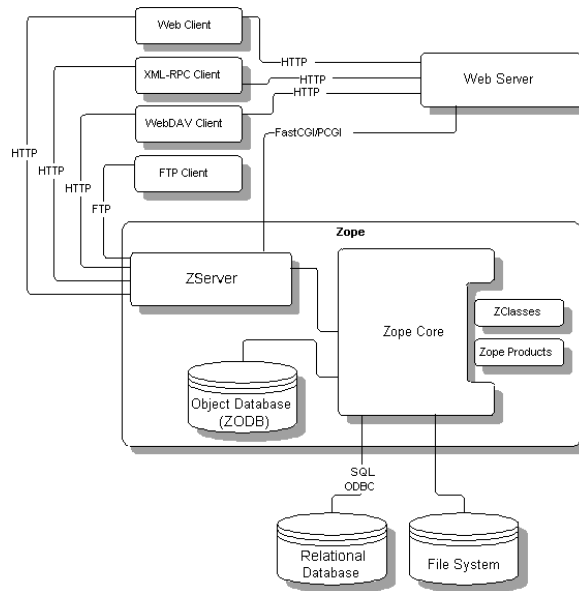


Figure 2-1 Zope Architecture

ZServer — Zope comes with a built in web server that serves content to you and your users. This web server also serves Zope content via FTP, WebDAV, and XML-RPC (a remote procedure call facility).

Web Server — 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. Zope works with these web servers also, and any other web server that supports the Common Gateway Interface (CGI).

Zope Core — This is the engine which coordinates the show, driving the management interface and object database.

Object Database — When you work with Zope, you are usually working with objects that are stored in Zope's object database.

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

File System — Zope can of course work with documents and other files stored on your server's file system.

ZClasses — Zope allows site managers to add new object types to Zope using the Zope Management Interface. ZClasses are these kinds of objects.

Products — Zope also allows site managers to add new object types to Zope by installing "Product" files on their Zope server's filesystem.

Installing and Starting Zope

By the end of this chapter you should be able to install and start Zope. It's fairly easy to install Zope on most platforms, and it should typically take you no longer than ten minutes.

Downloading Zope

Zope Corporation makes "binaries" which are available on Zope.org for the Windows, Linux and Solaris operating systems. These binaries are "ready-to-run" releases of the Zope application server that do not require compilation.

There are typically two types of Zope releases: a "stable" release and a "development" release. The "stable" Zope release is always available as a binary distribution for supported platforms. The "development" Zope release may or may not be distributed as a binary for any given platform. If you are new to Zope, you almost certainly want to use the "stable" Zope release.

You may download Zope from the Zope.org web site. The most recent stable and development versions are always available from the Download area of the Zope.org website.

For platforms for which there is no binary release, you must download the Zope source and compile it. Zope may be compiled on almost any Unix-like operating system. Zope has reportedly been successfully compiled on Linux, FreeBSD, NetBSD, OpenBSD, Mac OS X, HP-UX, IRIX, DEC OSF/1, and even Cygwin (the UNIX emulation platform for Windows). As a general rule of thumb, if Python is available for your operating system, and you have a C compiler and associated development utilities, then you can probably compile Zope. A notable exception is Mac OS 7/8/9. Zope does not run at all on these platforms.

Installing Zope

Zope requires different installation steps depending on your operating system platform. The sections below detail installing the binary version of Zope on Windows on Intel platforms, Solaris on SPARC platforms, and Linux on Intel platforms. We also detail a installation from source for platforms for which Zope Corporation does not provide a binary distribution.

Various binary Zope packages exist that are not distributed by Zope Corporation, but instead are distributed by third parties. Provided here is a list of URLs to these below for convenience's sake. These packages are not directly supported by Zope Corporation, although Zope Corporation encourages alternate binary distributions for unsupported platforms by third parties.

SPVI's Mac OS X binary distro

Jeff Rush's Zope RPMs for Linux

Adam Manock's Zope RPMs for Linux

FreeBSD Zope port

Debian Linux Zope package

Zope is also available from many Linux distributors as a "native" package. For example, RedHat often ships Zope on its "PowerTools" CD as an RPM. Check with your Linux operating system vendor to see if there are native Zope packages available for your platform.

Installing Zope for Windows With Binaries from Zope.org

The "Win32" version of Zope works under Windows 95, Windows 98, and Windows ME, Windows NT, Windows 2000, and Windows XP. Zope for Windows comes as a self-installing .exe file. To install Zope, first, download the Win32 executable installer from the Download area on Zope.org. It is typically named something like "Zope-2.X.X-win32-x86.exe" where the "X"s refer to the current Zope version number.

Important note: Do not try to use the file named "Zope-2.X.X-to-2.X.X-win32.x86.tgz" to install Zope for the first time. This is an upgrade package which upgrades an older version of Zope to a newer one instead of an installable Zope distribution.



The screenshot shows the Zope.org website interface. At the top, there is a navigation bar with links for "Zope Corp", "Search", "Download", "Docs", and "Re". Below this, the "Zope Community" logo is visible. On the left side, there is a sidebar menu with sections for "Guest" (Join Zope.org, Log in), "All Products" (Zope Hotfixes, Zope Releases, Zope RPMs), and "Zope Exits" (dev.zope.org, CMF Dogbowl, Zope Collector, Zope CVS, ZopeZen, Zope Newbies, Zope Labs, EuroZope, Zopera, FreeZope, NIP Free Zope, Hosting Zope). The main content area displays the following information:

- Created by [zopematt](#).
- Download [Zope-2.5.1-win32-x86.exe](#) (5501265 bytes)
- Date: 2002/04/23
- Version: 2.5.1 (Stable)
- Platform: win32, x86
- Contact: zope@zope.org
- [Changes](#)
- [License](#)
- [Installation Instructions](#)

Figure 2-1 Current stable Windows Zope Release

Download the current stable release installer for Windows from Zope.org using your web browser. Place the file in a temporary directory on your hard disk or on your Desktop. Once the installer file has been downloaded, navigate to the folder in which you downloaded the file to, and double-click on the file's icon. The installer then begins to walk you through the installation process.

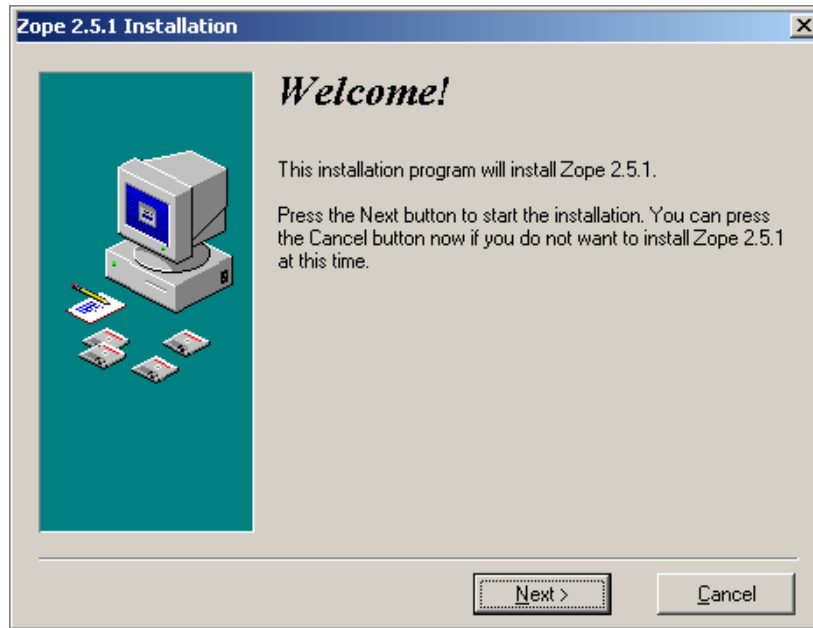


Figure 2-2 Beginning the installer

Click *Next* . You are asked to accept the Zope Public License before installing the product. After you read and accept the license, click *Next* again. Since you can install more than one Zope instance on on any given machine, you are asked to pick a unique "site name" for your Zope instance. The default name is "WebSite". It's recommended that you change this value. "Zope" is a reasonable name, although you are of course free to pick any name you choose.

Click *Next* after choosing your site's name. You are then asked to choose a directory in which to install Zope. A reasonable choice for a destination directory is "c:\Program Files\Zope". After filling in the directory name, click *Next* . You will be prompted to create a new Zope user account. This is not an operating system account. It is a user account that is only meaningful to Zope. The account that you specify is called the *initial user* (or "superuser") and is used to log into Zope for the first time. It is also given Zope administrative privileges. You can change this user name and password later if you wish. A reasonable choice for the initial user name is "admin".

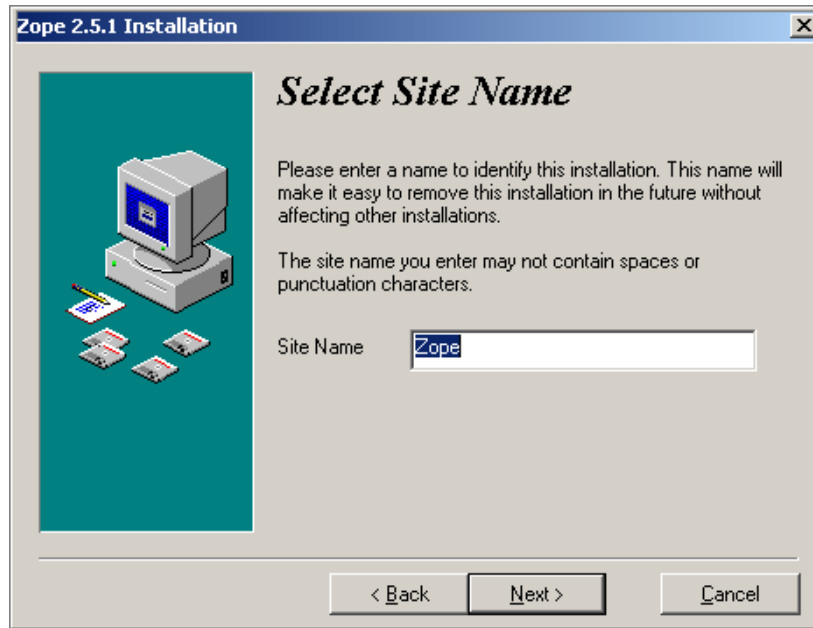


Figure 2-3 Selecting a Site Name

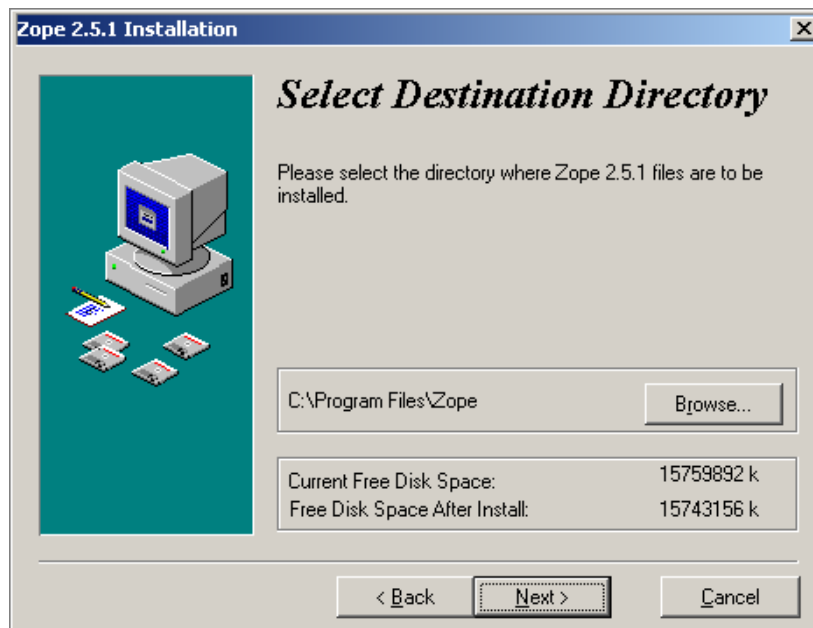


Figure 2-4 Selecting a Destination Directory

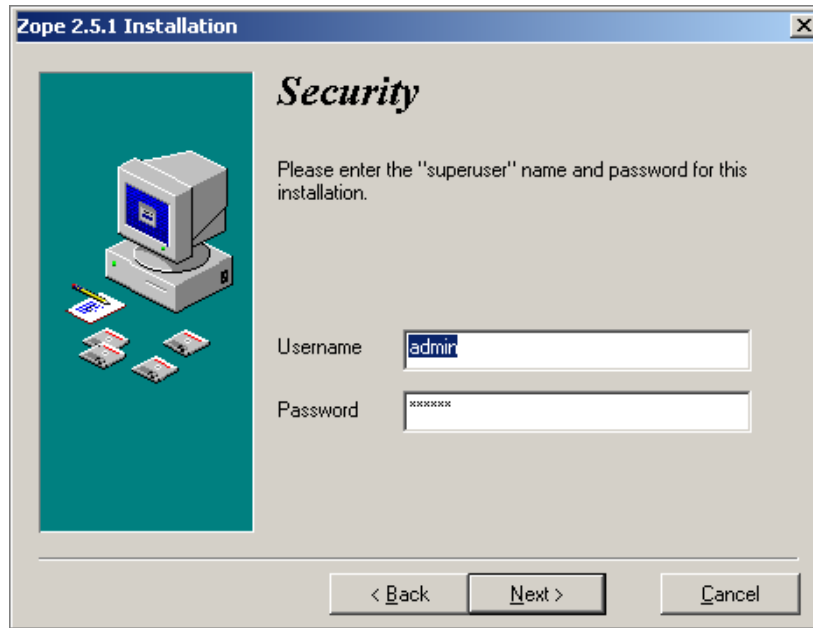


Figure 2-5 Provide an initial username and password

Click *Next* after choosing the initial user name and password. The installer presents a dialog indicating that it is ready to install files. Click *Next* again to begin installing the files.

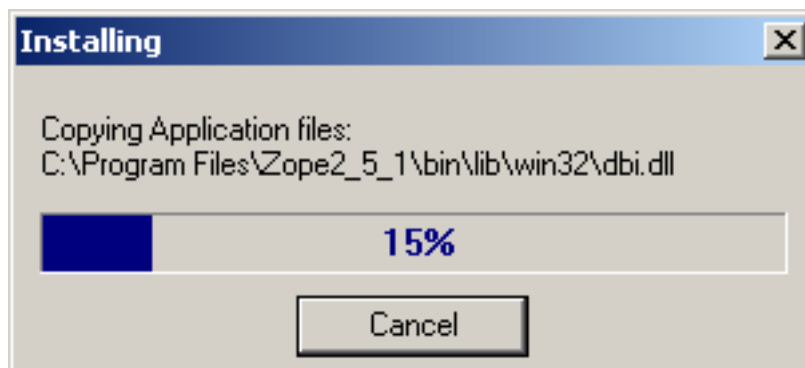


Figure 2-6 Installing files

Once the file copy is finished, if you are using Windows NT, Windows 2000, or Windows XP, you will see a dialog that indicates that you may choose to run Zope as a service. If you are just running Zope for personal use, don't bother running it as a service. If you are running Windows 95, Windows 98, or Windows ME, you cannot run Zope as a service (it is not offered as an option). It is recommended that if you are installing Zope for the first time that you don't choose to run the server manually.

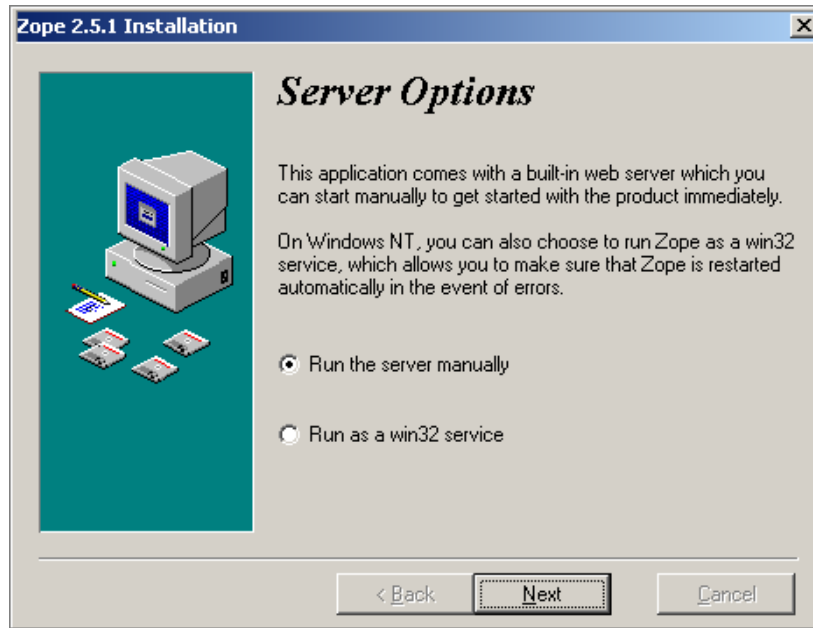


Figure 2-7 Server Options

After you click "Next", the installer informs you that the installation was successful. Click "Finish". If you decide to uninstall Zope later you can use the *Unwise.exe* program that resides in the directory in which you chose to install Zope.

Note that the Zope installer does not add a program folder entry to your "Start" menu. You will see how to start Zope in an upcoming section.

Installing Zope on Linux and Solaris With Binaries from Zope.org

The binary installations of Zope on Linux and Solaris are very similar. The binary distribution of Zope for Linux and Solaris comes as a *.tgz* file which must be uncompressed before you are able to begin the installation.

Important note: Do not try to use the file named "Zope-2.X.X-to-2.X.X-platform.tgz" to install Zope for the first time. This is an upgrade package which upgrades an older version of Zope to a newer one instead of an installable Zope distribution.

This paragraph has material that only applies to Solaris users. Before attempting to install Zope on Solaris for the first time, you need to install "GNUtar" and "gunzip". Both packages are available from the Solaris Package Archive . GNUtar is a "tape archive" program which, unlike the standard Solaris "tar" program is able to handle long file paths. Although Solaris comes with its own "tar" program, it is unable to handle unpacking Zope because it has a lame filepath-length limit that is exceeded by the length of some of the paths in the install package. "gunzip" is the GNU Lempel-Ziv encoding "unzip" program. Most, if not all, Linux versions come with GNUtar as the default "tar" program and already have gunzip installed, so if you run Linux, don't worry about this.

To begin a Zope installation, ownload the required installation archive from the Download area on Zope.org. It is typically named something like "Zope-2.X.X-solaris-sparc.tgz" (for Solaris) or "Zope-2.X.X-linux2-x86.tgz" (for Linux) where the "X"s refer to the current Zope version number.

After you download the installation archive for your platform, but before you install Zope, it is important that you decide where you'd like to install Zope and which user will be used to run it. It is suggested that Zope be unpacked and run as

a "normal" user (any user *except* the root user). Though you may of course create a "dedicated" Zope user account, we're going to assume you want to install it in a subdirectory of your own personal "home" directory for the purpose of these instructions.

Download the most recent stable binary installation archive for your platform into your user's "home" directory. Below we show a user using "wget" for this purpose, but you may download it via any web browser:

```
chrism@saints:~$ wget http://www.zope.org/Products/Zope/2.5.1/Zope-2.5.1-linux2-x86.tgz
--20:27:56-- http://www.zope.org:80/Products/Zope/2.5.1/Zope-2.5.1-linux2-x86.tgz
=> `Zope-2.5.1-linux2-x86.tgz.1'
Connecting to www.zope.org:80... connected!
HTTP request sent, awaiting response... 200 OK
Length: 5,979,458 [application/x-gzip]

  0K -> ..... [ 0%]
 50K -> ..... [ 1%]
(..and so on..)
```

Note that unlike most other UNIX programs, the Zope installer does not distinguish between a "build" directory and a "install" directory. The "build" directory *is* the "install" directory and vice versa. This means that the place where you unpack Zope and in which you run the installer should be the place where you want it to ultimately live. In our example case below, we're choosing to both unpack and install Zope into /home/chrism/Zope-2.5.1-linux2-x86.

"cd" to your home directory and, using gunzip and GNUtar, extract the files from the .tgz archive you downloaded in the last step:

```
chrism@saints:~$ gunzip -c Zope-2.5.1-linux2-x86.tgz | tar xvf -
Zope-2.5.1-linux2-x86/
Zope-2.5.1-linux2-x86/Extensions/
Zope-2.5.1-linux2-x86/Extensions/README.txt
Zope-2.5.1-linux2-x86/LICENSE.txt
Zope-2.5.1-linux2-x86/README.txt
(.. and so on..)
```

This will unpack Zope into a new directory named "Zope-2.X.X-osname-platformname" where the X's represent the current Zope version numbers, "osname" represents your OS name, and "platformname" represents your hardware platform name. "cd" to this Zope directory and run the Zope installer script. The command and output are shown below:

```
chrism@saints:~$ cd Zope-2.5.1-linux2-x86
chrism@saints:~/Zope-2.5.1-linux2-x86$ ./install
-----
Compiling python modules
-----
creating default inituser file
Note:
    The initial user name and password are 'admin'
    and 'tnLQ6imA'.

    You can change the name and password through the web
    interface or using the 'zpasswd.py' script.

chmod 0600 /home/chrism/Zope-2.5.1-linux2-x86/inituser
chmod 0711 /home/chrism/Zope-2.5.1-linux2-x86/var
-----
setting dir permissions
-----
creating default database
chmod 0600 /home/chrism/Zope-2.5.1-linux2-x86/var/Data.fs
-----
Writing the psgi resource file (ie cgi script), /home/chrism/Zope-2.5.1-linux2-x86/Zope.cgi
chmod 0755 /home/chrism/Zope-2.5.1-linux2-x86/Zope.cgi
-----
Creating start script, start
chmod 0711 /home/chrism/Zope-2.5.1-linux2-x86/start
-----
```

```
Creating stop script, stop
chmod 0711 /home/chrism/Zope-2.5.1-linux2-x86/stop
-----
```

```
Done!
chrism@saints:~/Zope-2.5.1-linux2-x86$
```

Note that the installer, among other things, will create an "initial" Zope user account with an autogenerated password. Write this username and password down temporarily. You will use this information to log in to Zope for the first time. You can change the initial user name and password later with the *zpasswd.py* script (see the chapter entitled Users and Security).

You have now successfully installed the Zope binary distribution. For more information on installing the binary distribution of Zope in alternate configurations on UNIX, see the installation instructions in the *INSTALL.txt* file inside the *doc* directory of the binary release package. You may additionally find out more about the installer script by running it with the *-h* (help) switch:

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

Compiling and Installing Zope from Source Code

If binaries aren't available for your platform, chances are good that you will be able to compile Zope from its source code. To do this, however, you first must:

- ensure you have a "C" compiler on your system (GNU gcc is preferred)
- ensure you have a recent "make" on your system (GNU make is preferred)
- install the Python language on your system from source.

Zope is written primarily in the Python language, and Zope requires Python to be able to run at all. Though binary versions of Zope ship with a recent Python, the source Zope distribution does not. Although we try to use the most recent Python for Zope, often the latest Python version is more recent than the version we "officially" support for Zope. For the most recent information on which version of Python you need to compile Zope with, see the release notes on the Web page for each version. Zope versions 2.5 and 2.6 require a Python 2.1 version equal to or greater than 2.1.3. Zope 2.3 and earlier versions require Python 1.5.2. **No version of Zope is yet officially compatible with any version of Python 2.2.**

You can obtain instructions for downloading, compiling and installing Python from source at the Python.org web site. Some Linux distributions ship with a preinstalled Python 2.1, but you need to be careful when attempting to use a vendor-installed Python to compile Zope. Some of these vendor-supplied Python distributions do not ship the necessary Python development files needed to compile Zope from source. Sometimes these development files are included in a separate "python-devel" package that you may install and use, but sometimes they are not. We recommend, to avoid headaches like this, that you compile and install Python from source if you wish to compile and install Zope from source.

After downloading, compiling, and installing Python from source, download the current Zope source distribution. See the Zope.org Downloads area for the latest Zope source release. Below we use "wget" for the purpose of downloading the source release, although you may of course use any browser or file retrieval utility:

```
chrism@saints:~$ wget http://www.zope.org/Products/Zope/2.5.1/Zope-2.5.1-src.tgz
--20:49:34-- http://www.zope.org:80/Products/Zope/2.5.1/Zope-2.5.1-src.tgz
=> `Zope-2.5.1-src.tgz'
Connecting to www.zope.org:80... connected!
HTTP request sent, awaiting response... 200 OK
Length: 2,165,141 [application/x-gzip]
OK -> ..... [ 2%]
```

The Zope Book (2.6 Edition)

```
50K -> ..... [ 4%]
100K -> ..... [ 7%]
```

(..and so on..)

Then extract the resulting `.tgz` archive into the place where you want Zope to be installed. Zope has no "build" directory, the "install" directory *is* the build directory. In the below example, we extract the `.tgz` directly into our home directory. This is recommended for purposes of this example:

```
chrism@saints:~$ gunzip -c Zope-2.5.1-src.tgz | tar xvf -
```

After extracting the `.tgz` file, "cd" to the resulting directory and, using the Python binary you compiled beforehand, invoke the Python script which compiles Zope. This script is cryptically named "wo_pcgi.py". "wo_pcgi" stands for "without PCGI", an artifact of Zope's web server integration roots, the meaning of which is largely unimportant today.:

```
chrism@saints:~$ cd Zope-2.5.1-src
chrism@saints:~/Zope-2.5.1-src$ python2.1 wo_pcgi.py
-----
Deleting '.pyc' and '.pyo' files recursively under /home/chrism/Zope-2.5.1-src...
Done.
```

```
-----
Compiling python modules
-----
```

```
Building extension modules
cp ./lib/python/Setup20 ./lib/python/Setup
```

```
-----
Compiling extensions in lib/python
cp /home/chrism/lib/python2.1/config/Makefile.pre.in .
make -f Makefile.pre.in boot PYTHON=
rm -f *.o *~
rm -f *.a tags TAGS config.c Makefile.pre python sedscrip
rm -f *.so *.sl so_locations
VERSION=`-c "import sys; print sys.version[:3]"`; \
```

(..and so on until...)

```
-----
creating default inituser file
```

Note:

```
The initial user name and password are 'admin'
and 'w!YzlsDT'.
```

```
You can change the name and password through the web
interface or using the 'zpasswd.py' script.
```

```
chmod 0600 /home/chrism/Zope-2.5.1-src/inituser
-----
```

Done!

You've now successfully installed Zope from source code. Note that the compile script, among other things, has created an "initial" Zope user account with an autogenerated password. Write this username and password down. You will use this information to log in to Zope for the first time. You can change the initial user name and password later with the `zpasswd.py` script (see the chapter entitled Users and Security). The initial user has "administrator" privileges within this Zope instance.

Starting Zope

Zope is managed via a web browser, and Zope contains its own web server (named "ZServer"). A successful Zope startup implies that its web server starts, allowing you to access the Zope management interface via your web browser. You can access Zope's management interface from the same machine on which Zope runs, or you can access it from a

remote machine that is connected to the same network as your Zope server.

Zope's ZServer will "listen" for HTTP (web browser, or Hypertext Transfer Protocol) requests on TCP port 8080. If your Zope instance fails to start, make sure you don't have another application running which is already using TCP port 8080.

Zope also has the capability to listen on other TCP ports. Zope supports separate TCP ports for FTP (File Transfer Protocol), "monitor" (internal debugging), WebDAV (Web Distributed Authoring and Versioning), and ICP (Internet Cache Protocol) access. If you see messages which indicate that Zope is listening on ports other than the default 8080 HTTP, don't panic, it's likely normal.

Using Zope With An Existing Webserver

If you wish, you can configure your existing web server to serve Zope content. Zope interfaces with Microsoft IIS, Apache, and other popular web servers.

The Virtual Hosting Services chapter of this book provides rudimentary setup information for configuring Zope behind Apache. However, configuring Zope for use behind an existing webserver can be a complicated task, and there is more than one way to get it done. In the interest of completeness, here are some additional resources which should get you started:

- Apache: see the excellent DevShed article entitled *Using Zope With Apache* .
- IIS: see brianh's *HowTo* on using IIS with Zope. Also of interest may be the `WEBSERVER.txt` file in your Zope installation's `doc` directory, and andym's *Zope Behind IIS HowTo* .

If you are just "getting started" with Zope, note that it is not necessary to configure Apache or IIS (or any other webserver) to serve your Zope pages, as Zope comes with its own webserver. You typically only need to configure your existing webserver if you want to use it to serve Zope pages in a production environment.

Starting Zope On Windows

If you installed Zope to "run manually" (as opposed to installing Zope as a "service"), use Windows Explorer to navigate to the directory into which you installed the Zope instance (typically `c:\Program Files\Zope` or `c:\Program Files\WebSite`). Within this directory, find a file called *start.bat* . Double-click the *start.bat* icon. A console window will be opened. It will display process startup information.

If chose to run Zope as a "service" on Windows NT/2000/XP, you can start Zope via the standard Windows "Services" control panel application. A Zope started as a service writes events to the standard Windows Event Log; you can keep track of when your service starts and stops by reviewing your system's Event Log. A Zope instance which has been installed as a "service" can also be run manually by invoking the *start.bat* file in the Zope installation directory as described above.

Starting Zope on UNIX

Important note: If you installed Zope from an RPM or a another "vendor distribution" instead of installing a Zope Corporation-distributed binary or source release, the instructions below may be not be applicable. Under these circumstances, please read the documentation supplied by the vendor to determine how to start your Zope instance instead of relying on the instructions below.

To start Zope, "cd" into to the directory in which you installed Zope and invoke the shell script named " *start* ". Here is an example of this invocation and its typical output:

The Zope Book (2.6 Edition)

```
chrism@saints:~$ cd Zope-2.5.1-linux2-x86
chrism@saints:~/Zope-2.5.1-linux2-x86$ ./start
-----
2002-06-28T03:17:02 INFO(0) ZODB Opening database for mounting: '142168464_1025234222.179125'
-----
2002-06-28T03:17:02 INFO(0) ZODB Mounted database '142168464_1025234222.179125' at /temp_folder
-----
2002-06-28T03:17:17 INFO(0) Zope New disk product detected, determining if we need to fix up any ZClasses.
-----
2002-06-28T03:17:17 INFO(0) ZServer HTTP server started at Thu Jun 27 23:17:17 2002
      Hostname: saints
      Port: 8080
-----
2002-06-28T03:17:17 INFO(0) ZServer FTP server started at Thu Jun 27 23:17:17 2002
      Hostname: saints
      Port: 8021
-----
2002-06-28T03:17:17 INFO(0) ZServer CGI Server started at Thu Jun 27 23:17:17 2002
      Unix socket: /home/chrism/Zope-2.5.1-linux2-x86/var/cgi.soc
```

Starting Zope As The Root User

ZServer (Zope's server) supports `setuid()` on POSIX systems in order to be able to listen on low ports such as 21 (FTP) and 80 (HTTP) but drop root privileges when running; on most POSIX systems only the `root` user can do this. Versions of Zope prior to 2.6 had less robust versions of this support. Several problems were corrected for the 2.6 release.

The most important thing to remember about this support is that you don't *have* to start ZServer as root unless you want to listen for requests on "low" ports. In fact, if you don't have this need, you are much better off just starting ZServer as a user account dedicated to running Zope. `nobody` is not a good idea for this user account; see below.

If you do need to have ZServer listening on low ports, you will need to start `z2.py` as the `root` user, and also specify what user ZServer should `setuid()` to. Do this by specifying the `-u` option followed by a username or UID, either in the `start` script or on the `z2.py` command line. The default used to be 'nobody'; however if any other daemon on a system that ran as `nobody` was compromised, this would have opened up your Zope object data to compromise.

You must also make sure the "var" directory is owned by root, and that it has the sticky bit set. This is done by the command `chmod o+t var` on most systems. When the sticky bit is set on a directory, anyone can write files, but nobody can delete others' files in order to rewrite them. This is necessary to keep others from overwriting the PID file, tricking root into killing processes when `stop` is run.

Your Zope Installation

To use and manage Zope, you'll need a web browser. Zope's management interface is written entirely in HTML, therefore any browser that understands modern HTML allows you to manage a Zope installation. Mozilla, and any 3.0+ version of Microsoft Internet Explorer or Netscape Navigator will do. Other browsers that are known to work with Zope include Opera, Galeon, Konqueror, OmniWeb, Lynx, and W3M.

Start a web browser on the same machine on which you installed Zope and visit the URL `http://localhost:8080/`. If your Zope is properly installed and you're visiting the correct URL, you will be presented with the Zope "QuickStart" screen.

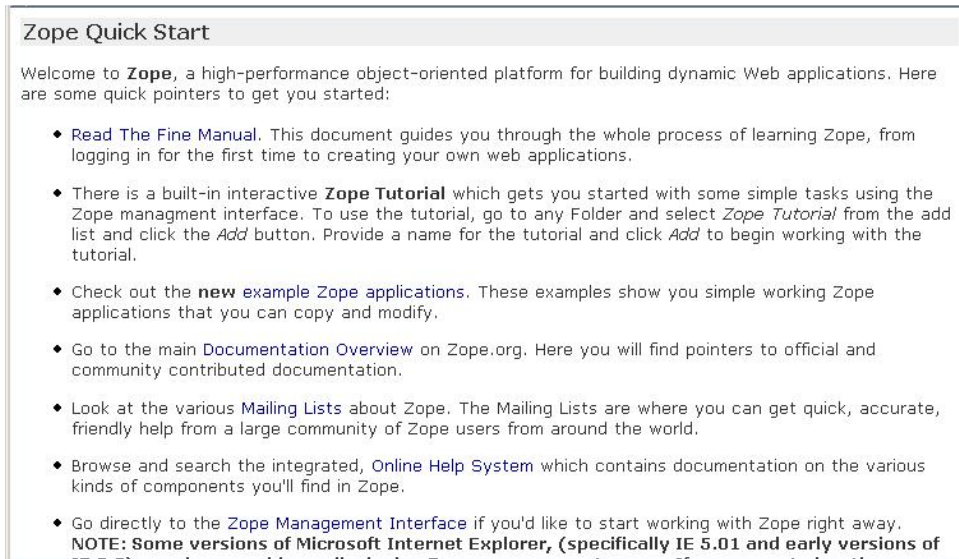


Figure 2-8 Zope QuickStart

If you see this screen, congratulations! You've installed Zope successfully. If you don't, see the *Troubleshooting* section below.

Logging In

To do anything remotely interesting with Zope, you need to use its "management interface". Zope is completely web-manageable. To log into the Zope management interface, use your web browser to navigate to Zope's management URL. Assuming you have Zope installed on the same machine from which you are running your web browser, the Zope management URL will be `http://localhost:8080/manage`.

Successful contact with Zope using this URL will result in an authentication dialog. In this dialog enter the "initial" username and password you chose when you installed Zope. You will be presented with the Zope Management Interface (ZMI).

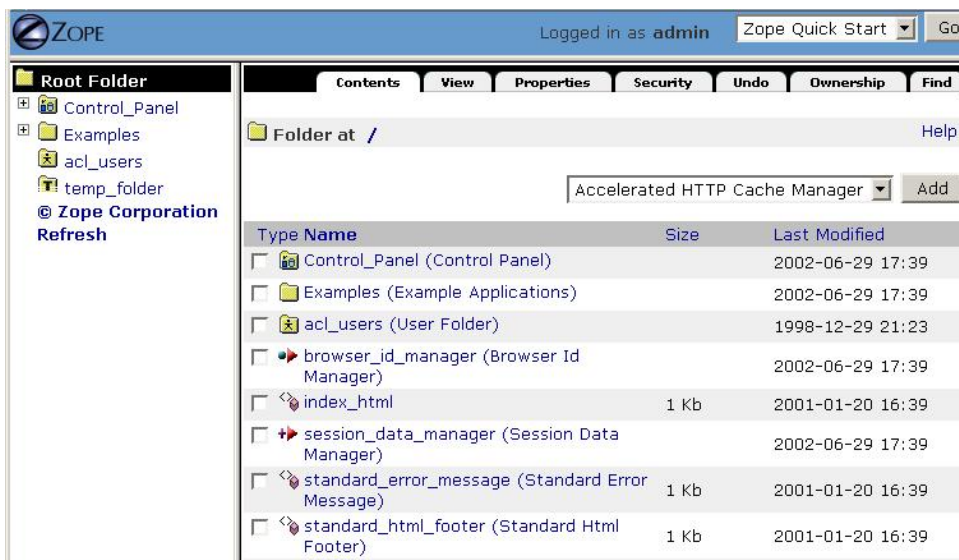


Figure 2-9 The Zope management interface.

If you do not see an authentication dialog and the Zope Management interface, refer to the *Troubleshooting* section of this chapter.

Controlling the Zope Process With the Control Panel

When you are using the ZMI, you can use the *Zope Control Panel* to control the Zope process. Find and click the `Control_Panel` object in ZMI.

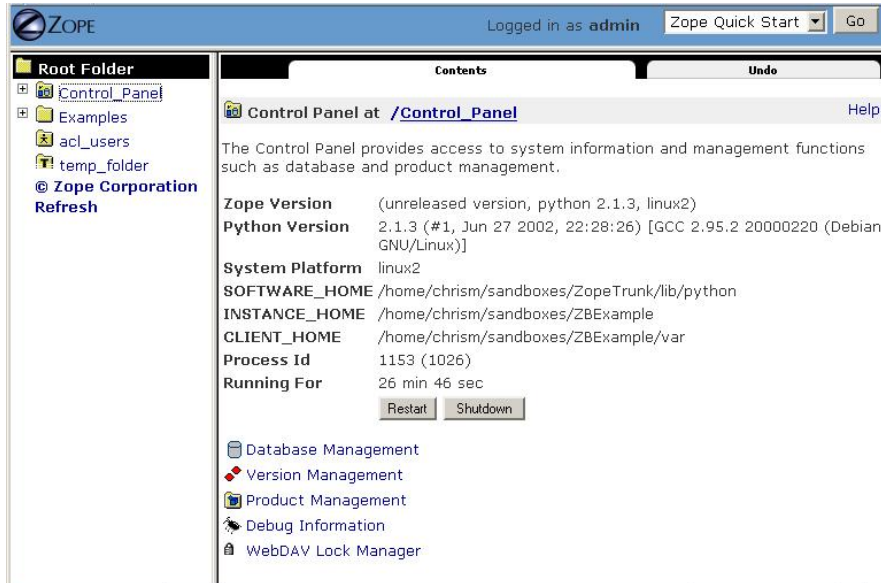


Figure 2-17 The Control Panel

The Control Panel displays information about your Zope, such as the Zope version you're running, the Python version that Zope is using, the system platform, the "SOFTWARE_HOME" (your Zope directory), the "INSTANCE_HOME" (typically the same as your zope home), your "CLIENT_HOME" directory (the "var" directory of your Zope), Zope's process id, and how long Zope has been running for. Several buttons and links will also be shown.

If you are running Zope on UNIX or as a service on Windows, you will see a button in the Control Panel named *Restart*. If you click the *Restart* button, Zope will shut down and then immediately start up again. It may take Zope a few seconds to come back up and start handling requests. You needn't shut your web browser down and restart it to resume using Zope after pressing *Restart*, just wait for the Control Panel display to reappear.

To shut Zope down from the ZMI, click the *Shutdown* button. Shutting Zope down will cause the server to stop handling requests and exit. You will have to manually start Zope to resume using it. Shut Zope down only if you are finished using it and you have the ability to access the server on which Zope is running, so that you can manually restart it later. If you see a "strange" message appear in your web browser when you shut Zope down, don't panic. This is normal. A normal shutdown presents the user with a web page that states:

```
An error was encountered while publishing this resource
exceptions.SystemExit
Zope has exited normally
( .. more output ..)
```

Controlling the Zope Process From the Command Line

To stop a manually-run Zope on Windows press "Ctrl-C" while the console window under which Zope is running is selected. To stop a Zope on Windows that was run as a service, find the service with the name you assigned to your Zope installed in the Services Control Panel application and stop the service.

To stop Zope on UNIX, press "Ctrl-C" in the terminal window from which you started Zope or use the UNIX "kill" command against the lowest-numbered Zope process id. Zope processes under UNIX will be listed in "ps" output as "python z2.py [options]". This process id can also be found in the "var/Z2.pid" file inside of your Zope directory.

Troubleshooting

If your browser fails to connect with anything on TCP port 8080, your Zope may be running on a nonstandard TCP port (for example, some versions of Debian Linux ship with Zope's TCP port as 9673). 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: 9673
-----
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 that Zope's web server is listening on port 9673. This means that the management URL is <http://peanut:9673/manage>.

Certain versions of Microsoft Internet Explorer 5.0.1 and 5.5 have issues with the Zope management interface which manifest themselves as an inability to properly log in. If you have troubles logging in with IE 5.0.1 or IE 5.5, try a different browser or upgrade to IE 6.

If you forget or lose the initial user name and password, shut Zope down and change the initial user password with the `zpasswd.py` script and restart Zope. See the chapter entitled Users and Security for more information about configuring the initial user account.

Options To The Zope `start` or `start.bat` Script

The Zope startup script named `start` (or `start.bat` on Windows) has many command-line switch options. They are the same for UNIX and Windows (although some only work on one or the other). These command-line switches are detailed below:

```
-h
    Output help text.

-z path
    The location of the Zope installation.
    The default is the location of the "z2.py" script.

-Z path
    Unix only! This option is ignored on windows.

    If this option is specified, a separate management process will
    be created that restarts Zope after a shutdown (or crash).
```

The Zope Book (2.6 Edition)

The path must point to a pid file that the process will record its process id in. The path may be relative, in which case it will be relative to the Zope location.

To prevent use of a separate management process, provide an empty string: `-Z=''`

`-t n`

The number of threads to use. The default is 4.

`-i n`

Set the interpreter check interval. This integer value determines how often the interpreter checks for periodic things such as thread switches and signal handlers. The Zope default is 500, but you may want to experiment with other values that may increase performance in your particular environment.

`-D`

Run in Zope debug mode. This causes the Zope process not to detach from the controlling terminal, and is equivalent to supplying the environment variable setting `Z_DEBUG_MODE=1`

`-a ipaddress`

The IP address to listen on. If this is an empty string (e.g. `-a ''`), then all addresses on the machine are used.

`-d ipaddress`

IP address of your DNS server. If this is an empty string (e.g. `-d ''`), then IP addresses will not be logged. If you have DNS service on your local machine then you can set this to 127.0.0.1.

`-u username or uid number`

The username to run Zope as. You may want to run Zope as a dedicated user. This only works under Unix. If Zope is started as root, it is a required parameter.

`-P [ipaddress:]number`

Set the web, ftp and monitor port numbers simultaneously as offsets from the number. The web port number will be `number+80`. The FTP port number will be `number+21`. The monitor port number will be `number+99`.

The number can be preceeded by an ip address followed by a colon to specify an address to listen on. This allows different servers to listen on different addresses.

Multiple `-P` options can be provided to run multiple sets of servers.

`-w port`

The Web server (HTTP) port. This defaults to 8080. If this is a dash (e.g. `-w -`), then HTTP is disabled.

The number can be preceeded by an ip address followed by a colon to specify an address to listen on. This allows different servers to listen on different addresses.

Multiple `-w` options can be provided to run multiple servers.

`-W port`

The "WebDAV source" port. If this is a dash (e.g. `-w -`), then "WebDAV source" is disabled. The default is disabled. Note that this feature is a workaround for the lack of "source-link" support

The Zope Book (2.6 Edition)

in standard WebDAV clients.

The port can be preceeded by an ip address follwed by a colon to specify an address to listen on. This allows different servers to listen on different addresses.

Multiple -W options can be provided to run multiple servers.

-C
--force-http-connection-close

If present, this option causes Zope to close all HTTP connections, regardless of the 'Connection:' header (or lack of one) sent by the client.

-f port

The FTP port. If this is a dash (e.g. -f -), then FTP is disabled. The standard port for FTP services is 21. The default is 8021.

The port can be preceeded by an ip address follwed by a colon to specify an address to listen on. This allows different servers to listen on different addresses.

Multiple -f options can be provided to run multiple servers.

-p path

Path to the PCGI resource file. The default value is var/pcgi.soc, relative to the Zope location. If this is a dash (-p -) or the file does not exist, then PCGI is disabled.

-F path_or_port

Either a port number (for inet sockets) or a path name (for unix domain sockets) for the FastCGI Server. If the flag and value are not specified then the FastCGI Server is disabled.

-m port

The secure monitor server port. If this is a dash (-m -), then the monitor server is disabled. The monitor server allows interactive Python style access to a running ZServer. To access the server see medusa/monitor_client.py or medusa/monitor_client_win32.py. The monitor server password is the same as the Zope emergency user password set in the 'access' file. The default is to not start up a monitor server.

The port can be preceeded by an ip address follwed by a colon to specify an address to listen on. This allows different servers to listen on different addresses.

Multiple -m options can be provided to run multiple servers.

--icp port

The ICP port. ICP can be used to distribute load between back-end zope servers, if you are using an ICP-aware front-end proxy such as Squid.

The port can be preceeded by an ip address follwed by a colon to specify an address to listen on. This allows different servers to listen on different addresses.

Multiple --icp options can be provided to run multiple servers.

-l path

Path to the ZServer log file. If this is a relative path then the log file will be written to the 'var' directory. The default is 'var/Z2.log'.

The Zope Book (2.6 Edition)

-r

Run ZServer in read-only mode. ZServer won't write anything to disk. No log files, no pid files, nothing. This means that you can't do a lot of stuff like use PCGI, and zdaemon. ZServer will log hits to STDOUT and zLOG will log to STDERR.

-L

Enable locale (internationalization) support. The value passed for this option should be the name of the locale to be used (see your operating system documentation for locale information specific to your system). If an empty string is passed for this option (-L ''), Zope will set the locale to the user's default setting (typically specified in the \$LANG environment variable). If your Python installation does not support the locale module, the requested locale is not supported by your system or an empty string was passed but no default locale can be found, an error will be raised and Zope will not start.

-X

Disable servers. This might be used to effectively disable all default server settings or previous server settings in the option list before providing new settings. For example to provide just a web server:

```
./start -X -w80
```

-M file

Save detailed logging information to the given file. This log includes separate entries for:

- The start of a request,
- The start of processing the request in an application thread,
- The start of response output, and
- The end of the request.

Environment Variables that Effect Zope at Runtime

Zope behavior is also effected by the presence and value of operating system environment variables that are available in the shell from which Zope is started.

To set an OS environment variable under UNIX in the `bash` shell, use the "export" command e.g. `export EVENT_LOG_FILE=/home/chris/Zope/var/event.log` . To set an OS environment variable under Windows NT/2000, use the Control Panel -> System applet or use the DOS-mode "set" command e.g. `set EVENT_LOG_FILE=c:\chris\Zope\var\event.log` . The "set" command can also be used in Windows 98/ME. Below are the environment variables that effect Zope runtime behavior, including descriptions of each:

Zope library paths

PYTHONPATH

Effects the library load path used by Python. See "The Python Tutorial Modules Chapter":<http://www.python.org/doc/current/tut/node8.html> for more information about PYTHONPATH.

INSTANCE_HOME

If an INSTANCE_HOME is defined and has a 'lib/python' sub directory, it will be added to the front of the PYTHONPATH. INSTANCE_HOME is usually used to separate the Zope core installation from application code and third-party modules/products.

See also: SOFTWARE_HOME

The Zope Book (2.6 Edition)

SOFTWARE_HOME

The SOFTWARE_HOME usually keeps the directory name of the Zope core installation.

See also: INSTANCE_HOME

ZOPE_HOME

ZOPE_HOME is the root of the Zope software, where the ZServer package, z2.py, and the default import directory may be found.

Profiling

PROFILE_PUBLISHER

If set, Zope is forced profile every request of the ZPublisher. The profiling information is written to the value of the PROFILE_PUBLISHER.

Access Rules and Site Roots

SUPPRESS_ACCESSRULE

If set, all SiteRoot behaviors are suppressed.

SUPPRESS_SITEROOT

If set, all access rules behaviors are suppressed.

ZEO-related

CLIENT_HOME

CLIENT_HOME allows ZEO clients to easily keep distinct pid and log files. This is currently an **experimental** feature.

ZEO_CLIENT

If you want a persistent client cache which retains cache contents across ClientStorage restarts, you need to define the environment variable, ZEO_CLIENT, to a unique name for the client. This is needed so that unique cache name files can be computed. Otherwise, the client cache is stored in temporary files which are removed when the ClientStorage shuts down.

Debugging and Logging

EVENT_LOG_FORMAT or STUPID_LOG_FORMAT

Set this variable if you like to customize the output format of Zope event logger. EVENT_LOG_FORMAT is the preferred envvar but STUPID_LOG_FORMAT also works.

EVENT_LOG_FILE="path" or STUPID_LOG_FILE="path"

The event file logger writes Zope logging information to a file. It is not very smart about it - it just dumps it to a file and the format is not very configurable - hence the name STUPID_LOG_FILE. EVENT_LOG_FILE is the preferred envvar but STUPID_LOG_FILE also works.

See also: LOGGING.txt in top-level Zope "doc" directory.

EVENT_LOG_SEVERITY <number> or STUPID_LOG_SEVERITY <number>

If set, Zope logs only messages whose severity is level is higher than the specified one. EVENT_LOG_SEVERITY is the preferred envvar but STUPID_LOG_SEVERITY also works.

The Zope Book (2.6 Edition)

ZSYSLOG="/dev/log"

Setting this environment variable will cause Zope to try and write the event log to the named UNIX domain socket (usually '/dev/log'). This will only work on UNIX.

See also: LOGGING.txt

ZSYSLOG_FACILITY="facilityname"

Setting this environment variable will cause Zope to use the syslog logger with the given facility. This environment variable is optional and overrides the default facility "user". This will only work on UNIX.

See also: LOGGING.txt in top-level Zope "doc" directory.

ZSYSLOG_SERVER="machine.name:port"

Setting this environment variable tells Zope to connect a UDP socket to machine.name (which can be a name or IP address) and 'port' which must be an integer. The default syslogd port is '514' but Zope does not pick a sane default, you must specify a port. This may change, so check back here in future Zope releases.

See also: LOGGING.txt in top-level Zope "doc" directory.

ZSYSLOG_ACCESS="/dev/log"

ZSYSLOG_ACCESS_FACILITY="facilityname"

ZSYSLOG_ACCESS_SERVER="machine.name:port"

Like ZSYSLOG, ZSYSLOG_FACILITY, and ZSYSLOG_SERVER, but controlling the sending of access information to syslog (rather than controlling the sending of the event log)

Z_DEBUG_MODE "yes" or "no"

BOBO_DEBUG_MODE "yes" or "no" (obsolete)

Run Zope in "debug mode" if set. Same as -D option to 'z2.py' or 'start'.

Misc.

Z_REALM "your realm"

BOBO_REALM "your realm" (obsolete)

Realm to be used when send HTTP authentication requests to a web client. The real string is displayed when the web browser pops up the username/password requester

Security related

ZOPE_SECURITY_POLICY

If this variable is set to "PYTHON", Zope will use the traditional Python based AccessControl implementation. By default and for performance reasons Zope will use the cAccessControl module.

ZSP_OWNEROUS_SKIP

If set, will cause the Zope Security Policy to skip checks relating to ownership, for servers on which ownership is not important.

ZSP_AUTHENTICATED_SKIP

If set, will cause the Zope Security Policy to skip checks relating to authentication, for servers which serve only anonymous content.

ZOPE_DTML_REQUEST_AUTOQUOTE

The Zope Book (2.6 Edition)

Set this variable to one of 'no', '0' or 'disabled' to disable autoquoting of implicitly retrieved REQUEST data that contain a '<' when used in a dtml-var construction. When *not* set to one of these values, all data implicitly taken from the REQUEST (as opposed to addressing REQUEST.varname directly), that contain a '<', will be HTML quoted when interpolated with a <dtml-var> or &dtml-; construct.

ZODB related

ZOPE_DATABASE_QUOTA

If this variable is set, it should be set to an integer number of bytes. Additions to the database are not allowed if the database size exceeds the quota.

ZOPE_READ_ONLY

If this variable is set, then the database is opened in read only mode. If this variable is set to a string parsable by DateTime.DateTime, then the database is opened read-only as of the time given. Note that changes made by another process after the database has been opened are not visible.

Session related

ZSESSION_ADD_NOTIFY

An optional full Zope path name of a callable object to be set as the "script to call on object addition" of the session_data transient object container created in temp_folder at startup.

ZSESSION_DEL_NOTIFY

An optional full Zope path name of a callable object to be set as the "script to call on object deletion" of the session_data transient object container created in temp_folder at startup.

ZSESSION_TIMEOUT_MINS

The number of minutes to be used as the "data object timeout" of the "/temp_folder/session_data" transient object container.

ZSESSION_OBJECT_LIMIT

The number of items to use as a "maximum number of subobjects" value of the "/temp_folder" session data transient object container.

WebDAV

WEBDAV_SOURCE_PORT_CLIENTS

Setting this variable enables the retrieval of the document source through the standard HTTP port instead of the WebDAV port. The value of this variable is a regular expression that is matched against the user-agent string of the client.

Example::

```
WEBDAV_SOURCE_PORT_CLIENTS="cadaver.*" enables retrieval
of the document source for the Cadaver WebDAV client
```

Structured Text

STX_DEFAULT_LEVEL

The Zope Book (2.6 Edition)

Set this variable to change the default level for <Hx> elements. The default level is 3.

Esoteric

`Z_MAX_STACK_SIZE`

This variable allows you to customize the size of the Zope stack used by the SecurityManager (default 100).

When All Else Fails

If there's a problem with your installation that you just can't seem to solve, don't despair. You have many places to turn for help, including the Zope maillists and the #zope IRC channel.

If you are new to open source software, please realize that, for the most part, participants in the various "free" Zope support forums are volunteers. Though they are typically friendly and helpful, they are not obligated to answer your questions. Therefore, it's in your own self-interest to exercise your best manners in these forums in order to get your problem resolved quickly.

The most reliable way to get installation help is to send a message to the general Zope maillist detailing your installation problem. For more information on the available Zope mailing lists, see the Resources section of Zope.org. Typically someone on the "zope@zope.org" list will be willing to help you solve the problem.

For even more immediate help, you may choose to visit the #zope channel on the OpenProjects IRC (Internet Relay Chat) network. See the OpenProjects website for more information on how to connect to the OpenProjects IRC network.

If you are truly desperate and under a time constraint that prohibits you from utilizing "free" support channels, Zope Corporation provides for-fee service contracts which you can use for Zope installation help. See Zope.com for more information about Zope Corporation service contracts.

Object Orientation

To make best use of Zope, you will need to have a grasp of the concept of *object orientation*. Object orientation is a software development pattern that is used in many programming languages (C++, Java, Python, Eiffel, Modula-2, others) and computer systems which simulate "real-world" behavior. It stipulates that you should design an application in terms of *objects*. This chapter provides a broad overview of the fundamentals of object orientation from the perspective of a Zope developer.

Objects

In an object-oriented system (such as Zope), your application is designed around *objects*. Objects are self-contained "bundles" of data and logic. It is easiest to describe them by comparing them to other programming concepts.

In a typical non-object-oriented application, you will have two things:

- Code. For example, you may have a bit of logic in the form of a Perl script in a typical CGI-based web application which sucks employee data from a database and displays a table to a user.
- Data. For example, you may have employee data stored in a database such as MySQL or Oracle that your code operates upon by reading or changing it. This data exists almost solely for the purposes of the code that operates upon it; it has almost no value without the code.

In a typical object-oriented application, however, you will have one thing, and one thing only:

- Objects. Objects are collections of code and data wrapped up together. For example, you may have an "Employee" object that represents an employee. It will contain data about the employee, such as a phone number, name, and address, much like the information that would be stored in a database like MySQL or Oracle. However, the object will also contain "logic" (code) that can manipulate and display this data.

In a non-object-oriented application, your data is separate from your code. But in an object oriented application, both your data and your code is stored in one or more objects, each of which represents a particular "thing". Objects can represent just about anything. In Zope, the *Control_Panel* is an object, Folders which you create are objects, even the Zope "root folder" is an object. When you use the Zope "add list" to create a new item in the Zope Management Interface, you are creating an object. People who extend Zope by creating *Products* define their own types of objects which are then entered in to the Zope "add list", allowing you to create objects from them. A product author might define a "Form" object or a "Weblog" object. Basically, anything which can be described using a noun can be modelled as an object.

Object-orientation as a programming methodology allows software developers to design and create programs in terms of "real-world" things like Folders, Control_Panels, Forms, and Employees instead of designing programs based around more "computerish" concepts like bits, streams, and integers. Instead of teaching the computer about our problem by descending to its basic vocabulary (bits and bytes), we use an abstraction to teach the computer about the problem in terms of a vocabulary which is more natural to humans. The core purpose of object orientation is to allow developers to create, to the largest extent possible, a system based on abstractions of the natural language of a computer (bits and bytes) into real-world things (Employees and Forms) that we can understand more quickly and more readily.

This idea of abstraction also encourages programmers to break up a larger problem by addressing the problem as smaller, more independent "sub-problems". This allows developers to define solutions in terms of these "sub-problems". When you design an application in terms of objects, the pieces which eventually come to define the

solution to all the "sub-problems" of a particular "big" problem are objects.

Attributes

An object's data is defined by its *attributes*. For example, an attribute of an Employee object might be named "phone_number". This attribute will likely contain a series of characters which represent the employee's phone number. Other attributes of an Employee object might be "first_name" and "last_name", which are respectively, series of characters which represent the employee's first name and the employee's last name. Another attribute of an employee object might be `title`, which would be a series of characters representing the employee's job description.

An object typically uses attributes to store elements that describe itself. For example, "phone_number", "first_name", "last_name" and "title" describe an employee in a particular way. It may help to think of the set of attributes belonging to an object as a sort of "mini-database" which contains information representing the "real-world thing" that the object is attempting to describe. The complete collection of attributes assigned to an object defines the object's *state*. When one or more of an object's attributes are modified, the object is said to have *changed its state*.

Special kinds of web-editable object attributes in Zope are sometimes referred to as *Properties*.

Methods

The set of actions which an object may perform is defined by its *methods*. Methods are code definitions attached to an object which typically perform an action based on the *attributes* belonging to the object on which the method is defined. For example, a method of an Employee object named "getFirstName" may return the value of the object's "first_name" attribute, while a method of an Employee object named "setFirstName" might *change* the value of the object's "first_name" attribute. The "getTitle" method of an Employee object may return "Vice President" or "Janitor".

Methods are similar to *functions* in procedural languages like C. The key difference between a method and a function is that a method is "bound" to (attached to) an object, so instead of operating solely on "external" data that is passed in to it via arguments, it may also operate on the attributes of the object to which is bound.

Some objects in Zope are actually called "methods". For example, there are *DTML Methods*, *SQL Methods*, and *External Methods*. This is because these objects are meant to be used in a "methodish" way. They are "bound" to their containing Folder object by default when called, and the logic that they contain typically makes reference to their containing Folder. *Script (Python)* objects in Zope act similarly through their concept of "Bindings".

Messages

In an object-oriented system, to do any useful work, an object is required to communicate with other objects in the same system. For example, it wouldn't be particularly useful to have a single Employee object just sitting around in "object-land" with no way to communicate with it. It would then just be as "dumb" as a regular old relational database row, just storing some data. We want the capability to ask the object to do something useful. More precisely, we want the capability for *other* objects to ask our Employee object to do something useful. For instance, if we create an object named "EmployeeSummary", which has the responsibility for collecting the names of all of our employees for later display, we want the EmployeeSummary object to be able to ask a set of Employee objects for their first and last names.

When one object communicates with another, it is said to send a *message* to another object. Messages are sent to objects by way of the object's *methods*. For example, our EmployeeSummary object may send a message to our Employee object by way of "calling" its "getFirstName" method. Our Employee object would receive the message and return the value of its "first_name" attribute. Messages are sent from one object to another when a "sender" object calls a method of a "receiver" object.

When you access a URL that "points to" a Zope object, you are almost always sending that Zope object a message. When you request a response from Zope by way of invoking a Zope URL with a web browser, the Zope object publisher receives the request from your browser. It then sends a Zope object a message on your browser's behalf by "calling a method" on the Zope object specified by the URL. The Zope object responds to the object publisher with a return value, and the object publisher returns the value to your browser.

Classes and Instances

A class defines an object's behavior and acts as a constructor for an object. When we talk about a "kind" of object, like an "Employee" object, we actually mean "objects constructed using the Employee class" or, more likely, just "objects of the Employee class". Most objects are members of a class.

It is typical to find many objects in a system that are essentially similar to one another save for the values of their attributes. For instance, you may have many Employee objects in your system, each with "first_name" and "last_name" attributes. The only difference between these Employee objects is the values contained within their attributes. For example, the "first_name" of one Employee object might be "Fred" while another might be "Jim". It is likely that each of these objects should be *members of the same class*.

A class is to an object as a set of blueprints is to a house. Many houses can be constructed using the same set of blueprints; likewise many objects can be constructed using the same class. Objects that share a class typically behave identically to each other. If you visit two houses that share the same set of blueprints, you will likely notice striking similarities: the layout will be the same, the light switches will probably be in the same place, and the fireplace will almost certainly be in the same location. The shower curtains might be different in each house, but this is an *attribute* of each particular house which doesn't change its essential similarity with the other. It is much the same with instances of a class. If you "visit" two instance of a class, you will interact with both instances in essentially the same way: by calling the same set of methods on each. The data kept in the instance (by way of its attributes) might be different, but these instances *behave* in the same way.

The behavior of two objects constructed from the same class is similar because they both share the same *methods*. Methods of an object are not typically defined by the object itself, but instead are defined by the object's *class*. For instance, the Employee *class* defines the `getFirstName` method, and all objects that are members of that class share that method definition. The set of methods that are assigned to a class define the *behavior* of an object.

The objects which are constructed by a class are called *instances of the class* or (more often) just *instances*. For example, the Zope `Examples` folder is an *instance of the Folder class*. The `Examples` folder has an `id` attribute of `Examples`, while another folder may have an `id` attribute of `MyFolder`, but they are both instances of the same class, and behave identically. All of the objects that you deal with using the Zope management interface are instances of a class. Typically, the classes from which these objects are constructed are defined in Zope *Products*, which are created by Zope developers and community members.

Inheritance

Sometimes it is desirable for objects to share the same essential behavior, except for small deviations from each other. For example, you may want a `ContractedEmployee` object to have all the behavior of a "normal" `Employee` object except that you must keep track of a tax identification number on instances of the `ContractedEmployee` class that is irrelevant for "normal" instances of the `Employee` class.

Inheritance is the mechanism that allows you to share essential behavior between two objects, while customizing one with a slightly modified set of behaviors that differ from or extend the other.

Inheritance is specified at the *class level*. As we learned above, *classes define behavior*, and if we want to change object behavior, we almost always need to change its class.

If we base our "ContractedEmployee" class on the Employee class, but add a method to it named "getTaxIdNumber" and an attribute named "tax_id_number", the ContractedEmployee class would be said to *inherit from* the Employee class. In the jargon of object orientation, the ContractedEmployee class would be said to *subclass from* the Employee class and the *Employee* class would be said to be a *superclass of* the ContractedEmployee class.

When a subclass inherits behavior from another class, it doesn't need to sit by and accept the method definitions of its superclass. It can *override* the method definitions of its superclass. For instance, we may want to cause our ContractedEmployee class to return a different "title" than instances of our Employee class. In our ContractedEmployee class, we might cause the `getTitle` method of the Employee class to be *overridden* by creating a method within ContractedEmployee which has a different implementation. For example, it may always return "Contractor" instead of a job-specific title.

In Zope, inheritance is used extensively. For example, the Zope "Image" class inherits its behavior from the Zope "File" class, because images are really just another kind of file, and they share many behavior requirements. But the "Image" class adds a bit of behavior which allows it to "render itself" by printing an HTML tag instead of causing a file download. It does this by *overriding* the `index_html` method of the File class.

Object Lifetimes

Object instances have a specific *lifetime*. This lifetime is controlled typically controlled by either a programmer or a user of the system in which the objects "live".

Instances of web-manageable objects in Zope like Files, Folders, DTML Methods, and such have a lifetime of "from when a user creates them until he or she deletes them." You will often hear these kinds of objects described as *persistent* objects. These objects are stored in Zope's object database (the ZODB).

Other object instances have different lifetimes. There are object instances in Zope which last for a "programmer-controlled" period of time. For instance, the object that represents a web request in Zope (often called REQUEST), has a well-defined lifetime. Its lifetime lasts from the moment that the object publisher receives the request from a remote browser until the response is sent back to that browser. It is then destroyed automatically. Zope "session data" objects have another well-defined lifetime. These objects last from the time that a programmer creates one on behalf of the user via his code until such time that the system (on behalf of the programmer or site administrator) deems it necessary to throw away the object in order to conserve space or indicate an "end" to the user's session. This is defined by default as 20 minutes of "inactivity" by the user for whom the object was created.

Summary

Zope is an object-oriented development environment. Understanding Zope fully requires that you grasp the basic concepts of object orientation. You should attempt to understand attributes, methods, classes, and inheritance before setting out on a "for-production" Zope development project.

For a more lighthearted description of what object orientation is and how it relates to Zope, see Chris McDonough's *Gain Zope Enlightenment by Grokking Object Orientation*. For a more comprehensive treatment on the subject of object orientation, buy and read *The Object Primer* by Scott Ambler. There are also excellent object orientation tutorials available on the Internet. See *The Essence of Objects* chapter of the book "The Essence of Object Oriented Programming with Java and UML". There is an extensive Object FAQ available at Cyberdyne Object Systems.

Using The Zope Management Interface

Introduction

When you log in to Zope, you are presented with the Zope Management Interface (ZMI). The ZMI is a management and development environment that allows you to control Zope, manipulate Zope objects, and develop web applications.

The Zope Management Interface represents a view into the Zope *object hierarchy*. Almost every link or button in the ZMI represents an action that is taken against an *object*. When you build web applications with Zope, you typically spend most of your time creating and managing objects.

Don't be frightened if you don't understand the word "object" just yet. For the purposes of this chapter, the definition of an "object" is any discrete item that is manageable through the Zope Management Interface. In fact, for the purposes of this chapter, you can safely mentally replace the word "object" with the word "thing" with no ill effect. If you get confused, however, you may want to review the Object Orientation chapter for more background on objects.

How The Zope Management Interface Relates to Objects

Unlike a webserver like Apache or Microsoft IIS, Zope does not "serve up" HTML files that it finds on your server's hard drive. The objects that Zope creates are not stored in files that have an ".html" extension on your server's hard drive. There is no file hierarchy on your server's computer that contains all of your Zope objects.

Instead, the objects that Zope creates are stored in a database known as the ZODB, which stands for (unsurprisingly) the "Zope Object DataBase". "Out of the box", the ZODB creates a file named "Data.fs" in which Zope stores its objects. The Zope Management Interface is the primary way that you interact with Zope objects that are stored in this database. Note that there are other methods of interacting with objects stored in the ZODB, including FTP and WebDAV, which are detailed in the chapter in this book entitled *Managing Zope Using External Tools*, but the ZMI is the primary management tool.

ZMI Frames

The ZMI uses three browser frames. The left frame is called the *Navigator Frame*, and using it you may expand and contract a view into the Zope object hierarchy much like you would expand and contract a view of files using a file tree widget like the one in Windows Explorer. The right frame is called the *Workspace Frame*, and it displays a particular view of the object you're currently managing. The top frame is called the *Status Frame*, and it displays the name of the user who you are currently logged in as as well as a select list that allows you to perform various actions. We'll look more closely at these frames next.

The Navigator Frame

The left frame is the Navigator. In this frame you have a view into 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 "topmost" container of Zope objects. Almost everything meaningful in your Zope instance lives inside the root folder.

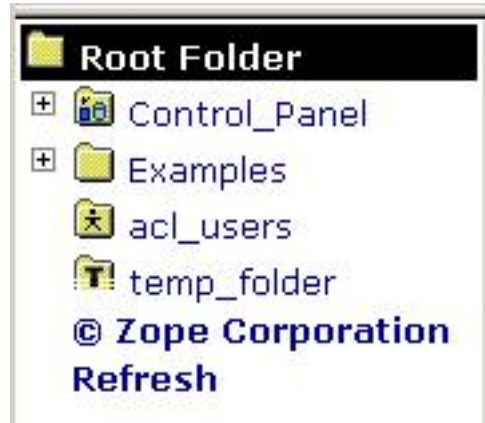


Figure 3-1 The Navigator Frame

Some of the folders in the Navigator are displayed with "plus mark" icons to their left. These icons let you expand the folders to see the sub-folders that are inside.

When you click on an object icon or name in the Navigator, the *Workspace* frame will refresh with a view of that object.

The Workspace Frame

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. The workspace gives you information about the current object, and lets you manage it.

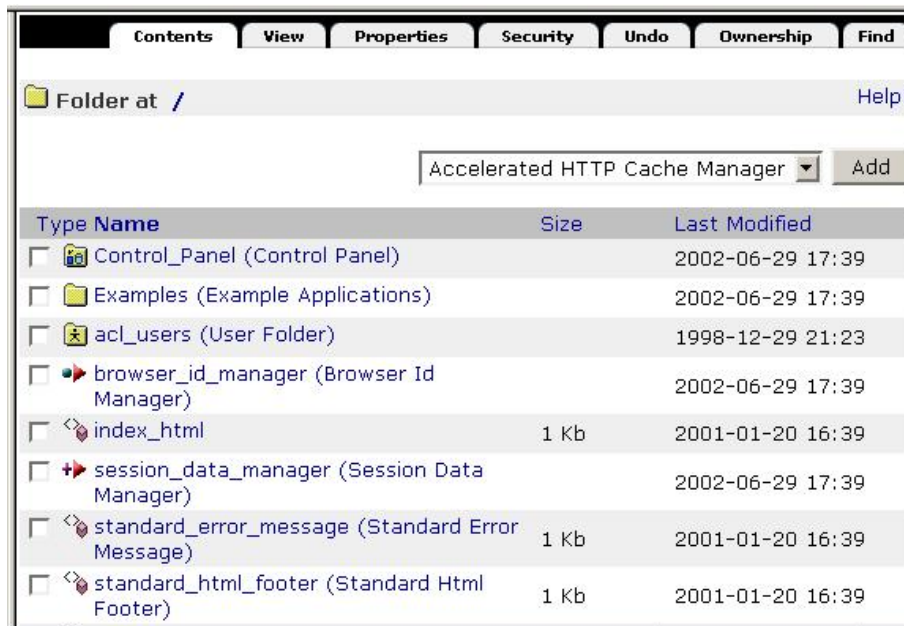


Figure 3-2 The Workspace Frame

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.

When you first log in to Zope, 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.

At the top of the page, `Folder at /` tells you that the current object is a folder and that its path is `/`. Note that this path is the object's place relative to Zope's "root" folder. The root folder's path is expressed as `/`, and since you are looking at the root when you first log in, the path displayed at the top of the workspace is simply `/`.

Zope object paths are typically mirrored in the URLs that you use to access a Zope object. For instance, if the main URL of your Zope site was `http://mysite.example.com:8080`, then the URL of the root folder would be `http://mysite.example.com:8080/` and the URL of `Folder at /myFolder` would be `http://mysite.example.com:8080/myFolder`.

As you explore different Zope objects, you find that the links displayed at the top of the workspace frame can be used to navigate between objects' management views. 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.

The Status Frame

In the "status 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 — By selecting this menu item, you can set default preferences for your Zope management interface experience. You can choose to turn off the status frame. You can also choose whether or not you want the management interface to try to use style sheets. Additionally, you can change the default height and width of textareas displayed in the ZMI. This information is associated with your browser via a cookie. It is not associated in any way with your Zope user account.

Logout — Selecting this menu item will log you out of Zope. Due to the way that the HTTP "basic authentication" protocol works, this may not work properly on all browsers. If you experience problems logging out using this facility, try closing and reopening your browser to log out.

Quick Start Links — Selecting this menu item presents the "QuickStart" page which has links to Zope documentation and community resources.



Figure 3-3 The Status Frame

Creating Objects

The Zope Management Interface allows you to create new objects in your Zope instance. To add a new object, select an entry from the pull-down menu in the Workspace labeled "Select type to add...". This pull-down menu is called the *add list*.

The first kind of object you'll want to add in order to "try out" Zope is a "Folder". To create a Zope Folder object, navigate to the root folder and select *Folder* from the add list. At this point, you'll be taken to an add form that collects information about the new folder, as shown in the figure below.

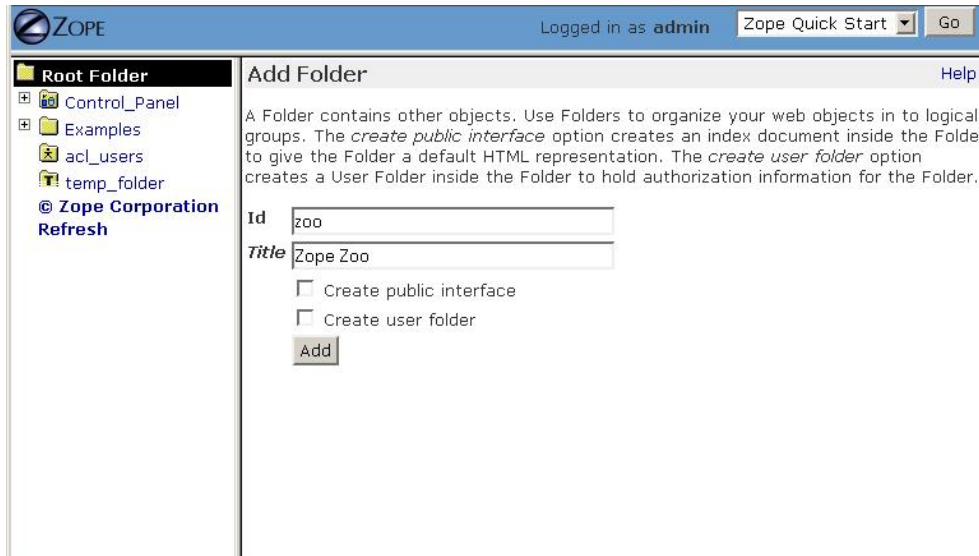


Figure 3-4 Folder add form.

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

Zope will create a new Folder object in the current folder named *zoo* . 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. The Workspace frame will switch to the contents view of *zoo* (which is currently an "empty" folder: it has no subobjects). Notice that the URL of the *zoo* 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* . Enter 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.

When you use Zope, you create new objects by following these steps:

1. Enter to the folder where you want to add a new object.
2. Choose the type of object you want to add from the add list.
3. Fill out the resulting add form and submit it.
4. Zope will create a new object in the 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* as a part of their *URL* . The URL of any given Zope object is typically a URL consisting of the folders in which the object lives plus its name. For example, we created a folder named "zoo" in the root folder. Its URL consists of "http://your.server.name/zoo" (where "your.server.name" is your server's name).

Moving and Renaming 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.

NOTE: Zope move and rename options require that you have cookies enabled in your browser.

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 selects the selected objects from the folder and places them on Zope's "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. Now, click the *Paste* button to paste cut object(s) into the *zoo* 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 removed. 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 root folder copy the *zoo* folder. Now paste it into the root folder. You will now have two folders inside the root folder, *zoo* and *copy_of_zoo* . 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_zoo* 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.

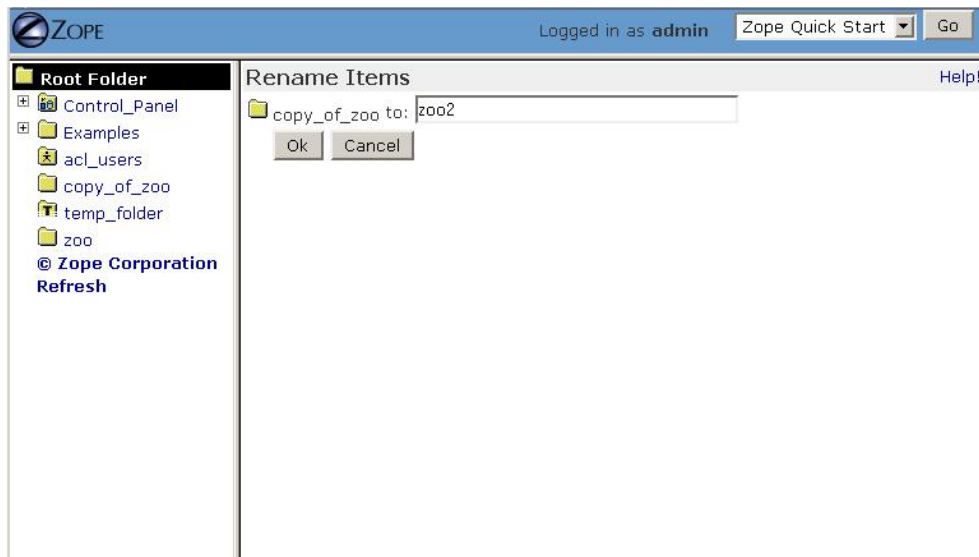


Figure 3-5 Renaming an Object

Type in a new id "zoo2" 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 root folder contains a *zoo* and a *zoo2* folder. Each of these folders contains a *bears* folder. This is because when we made a copy of the *zoo* folder it also copied the *bears* folder that it contained. Copying an object also copies all of the objects it contains.

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 using *Undo*.

Zope will not let you cut, delete, or rename a few particular objects in the root folder. These objects include *Control_Panel*, *browser_id_manager*, and *temp_folder*. These objects are necessary for Zope's operation. It is possible to delete other root objects, such as *index_html*, *session_data_manager*, *standard_html_header*, *standard_html_footer*, *standard_error_message*, and *standard_template.pt* but it is not recommended unless you have a good reason to do so.

Transactions and Undoing Mistakes

All objects you create in Zope are stored in Zope's "object database". Unlike other web application servers, Zope doesn't store its objects in files on a filesystem. Instead, all Zope objects are stored by default in a single special file on the filesystem named `Data.fs`. This file is stored in the `var` directory of your Zope instance. Using an object database rather than using file system files allows operations to Zope objects to be *transactional*.

A transactional operation is one in which all changes to a set of objects are committed as a single "batch". In Zope, a single web request initiates a transaction. When the web request is finished, Zope commits the transaction unless there was an error during the processing of the request. If there was an error, Zope refrains from committing the transaction. Each transaction describes all of the changes that happen in the course of performing a web request.

Any action in Zope that causes a transaction can be undone, via the *Undo* tab. You can recover from mistakes by undoing the transaction that represents the mistake.

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 the figure below.

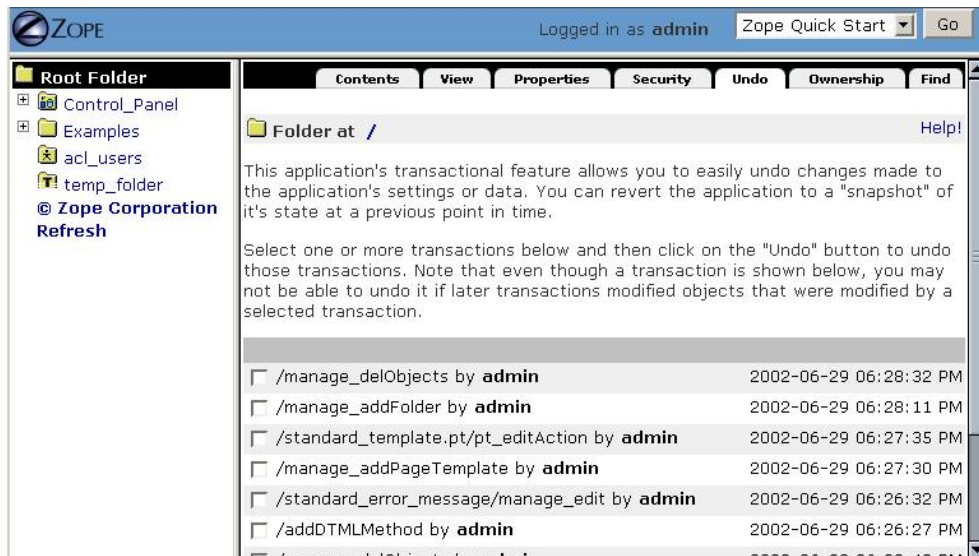


Figure 3-6 The Undo view.

Transactions are named after the Zope action (also known as a "method") that initiated them. In this case, the initiating action was one named `/manage_delObjects` (the action which deletes a Zope object).

Select the first transaction labeled `/manage_delObjects`, and click the *Undo* button at the bottom of the form, instructing Zope to undo the last transaction. You can verify that the task has been completed by visiting the root folder

to make sure that the `zoo` folder has returned. You may need to refresh your browser window to see the effect if you just use the "Back" button to revisit the root folder. To see the effect in the *Navigator* pane, click the "Refresh" link within the pane.

You may undo an undo (or in other words, perform a "redo"). You can undo and redo as many times as you like. When you perform a "redo", Zope inserts a transaction into the undo log describing the redo.

The Undo tab is available on most Zope objects. When viewing the Undo tab of a particular object, the list of undoable transactions is filtered down to the transactions that have recently effected the current object and its subobjects.

Undo Details and Gotchas

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*.

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.

Reviewing Change History

The Undo tab will provide you with enough information to know that a change has occurred. It, however, will not tell you much about the effect of the transaction on the objects that were changed during the transaction. "Presentation" and "logic" objects like DTML Methods, DTML Documents, Zope Page Templates, and Script (Python) objects support *History* for this purpose. If you know a transaction has effected one of these objects, you can go to that object's *History* View and look at the previous states of the object, as shown in the figure below.

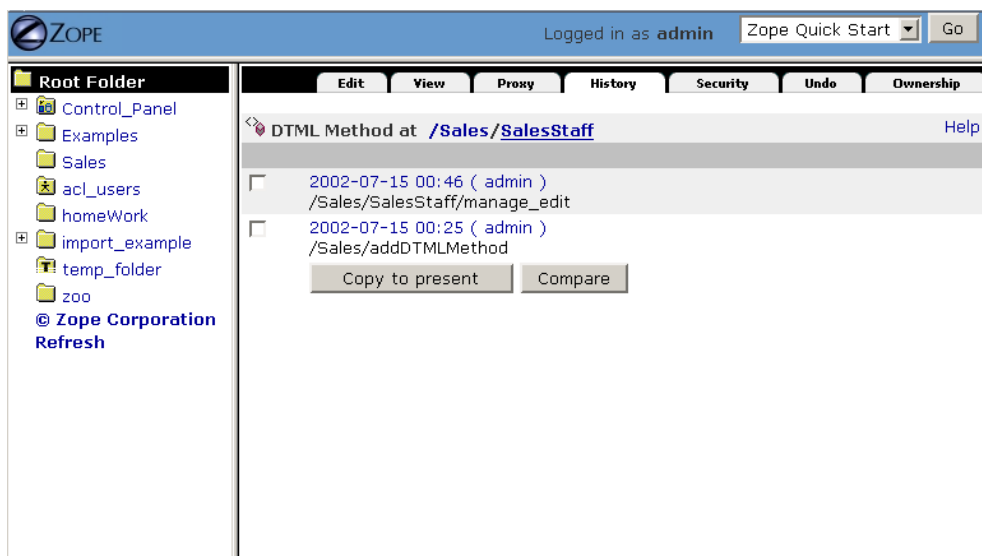


Figure 3-7 The History View

The *History* view of an object supports comparison of revisions, allowing you to track their changes over time. You may select two revisions from an object's History and compare them to one another. To perform a comparison between two

object revisions, select two revisions using the checkboxes next to the transaction identifiers, and click the *Compare* button.

The resulting comparison format is often called a *diff*. The diff format 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).

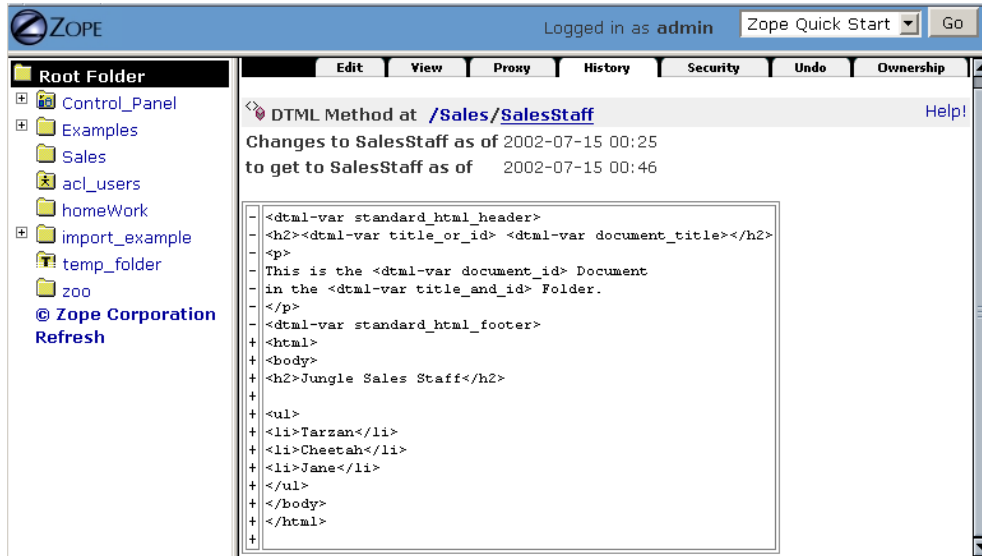


Figure 3-8 Comparing Revisions Via The History View

To revert to an older object revision, click the checkbox next to the transaction identifier, then click the *Copy to present* button.

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.

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. Create a folder in your root folder called "homeWork". After creating the folder, click the checkbox next to the *homeWork* folder you just created. Then click the *Import/Export* button. You should now be working in the Import/Export folder view, as shown in be figure below.

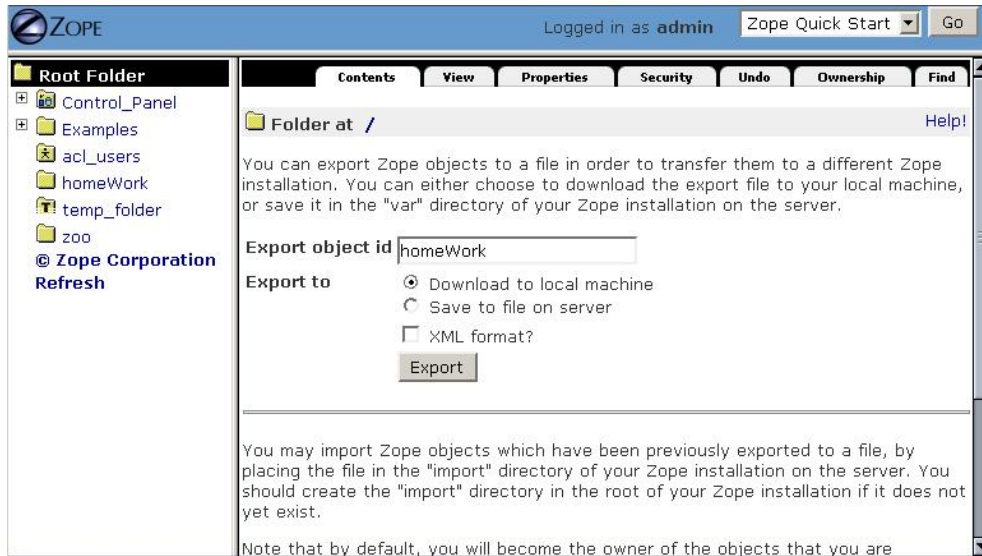


Figure 3-9 Exporting homeWork.zexp

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 file extension *.zexp*.

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. XML format exports are is much larger but are (mostly) human-readable. 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 export format looks like and want to examine it by hand.

While you're viewing the export form for *homeWork*, Ensure "download to local machine" is selected, "XML format?" is *not* selected, and click the *Export* button. Your browser will present a file save dialog. Save the file to a temporary location on your local computer (it will be named *homeWork.zexp*).

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 Zope server's filesystem. Here is an example of doing so. We are copying the *homeWork.zexp* file that's in a directory named */tmp* on the local computer to a remote ("home") computer running Zope using the *scp* facility on Linux. We copy the *.zexp* file into our Zope directory's "import" directory. In this example, the Zope installation directory on the remote computer is named */home/chrism/sandboxes/ZBExample*:

```
chrism@saints:/tmp$ ls -al homeWork.zexp
-rw-r--r--  1 chrism  chrism      182 Jul 13 15:44 homeWork.zexp
chrism@saints:/tmp$ scp homeWork.zexp saints.homeunix.com:/home/chrism/sandboxes/ZBExample/import
```

```
chrism@saints.homeunix.com's password:
homeWork.zexp 100% |*****| 182 00:00
chrism@saints:/tmp$
```

In the above example, the export file was copied from the local computer's /tmp directory to the remote computer's /home/chrism/sandboxes/ZBExample/import/homeWork.zexp file. Your local directory and your Zope's installation directory will be different. For purposes of this example, copy the export file you downloaded to your Zope installation's "import" directory using whatever facility you're most comfortable with (you needn't use scp).

Now, go back to your Zope's management interface. Create a Folder named import_example . Visit the newly-created import_example folder by clicking on it in the management interface. Then click on *Import/Export* button in the import_example folder and scroll to the bottom of the Workspace frame. Note that Zope gives you the option to either *Take ownership of imported objects* or *Retain existing ownership information* . Ownership will be discussed more in the chapter entitled "Users and Security". For now, just leave the *Take ownership of imported objects* option selected and, enter the name of the export file (homeWork.zexp) in the *Import file name* form element and click *Import* .

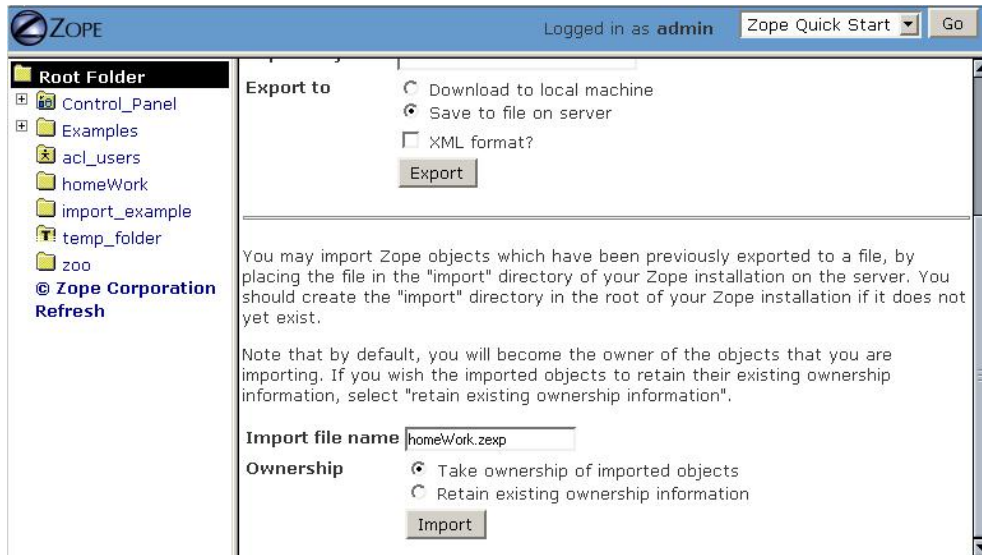


Figure 3-10 Importing homeWork.zexp

After you've clicked import, you will have a new object in the import_example folder named homeWork . Note that Zope informs you of the success of the import in a status message.



Figure 3-11 Success Importing homeWork.zexp

There are a few caveats to importing and exporting. In order to successfully perform an import of a Zope export file, you need to make sure that the Zope into which you're importing has the same *Products* installed. If an import fails, it's likely that you don't have the same Products installed in your Zope as the Products installed in the Zope from whence the export file came. Our example above works because we are exporting an object which is very common and which comes with all Zopes (a Folder). Check with the distributor of the export file to see what Products are necessary for proper import if you have problems importing a given export file.

Note that you cannot import an object into a folder that has an existing object with the same *id*. Therefore, when you import an export file, you need to ensure it does not want to install an object that has the same name as an existing object in the folder in which you wish to import it. In our example above, in order to bring your homework back to school, you'll either need to import it into a folder that doesn't already have a *homeWork* folder in it, or you'll need to delete the existing *homeWork* folder before importing the new one.

Using Object Properties

Properties are ways of associating information with objects in Zope. Many Zope objects, including folders, support properties. Properties can label an object in order to identify its contents. For example, 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 the *Properties* tab of the "root" object and you will be taken to the properties management view, as seen in the figure below.

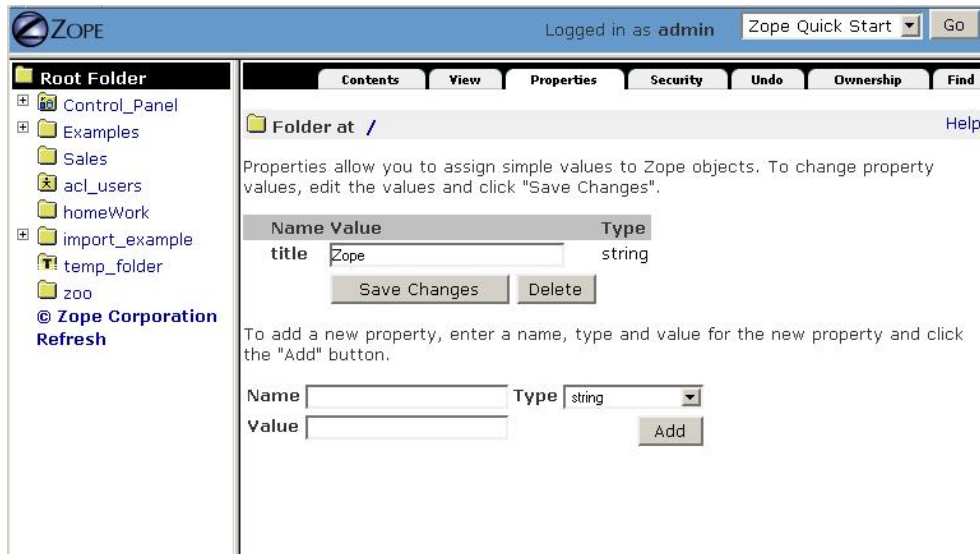


Figure 3-12 The Properties Management View

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 the figure above, you can see that the folder has a single string property: *title*. It has the value `Zope`. You may change any predefined property by changing its value in the Value box, clicking *Save Changes* afterwards. You may add additional properties to an object by entering a name, value, and type into the bottommost form on the Properties view the.

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 "metadata". Properties are supported by most Zope objects, and are often referenced by "dynamic" Zope objects such as "scripts" and "methods" (which we have not yet discussed) for purposes of data display.

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 which exposes the Zope Help System.

To see the help system, go to the root folder and click the *Help* link to the top far right in the Workspace frame.

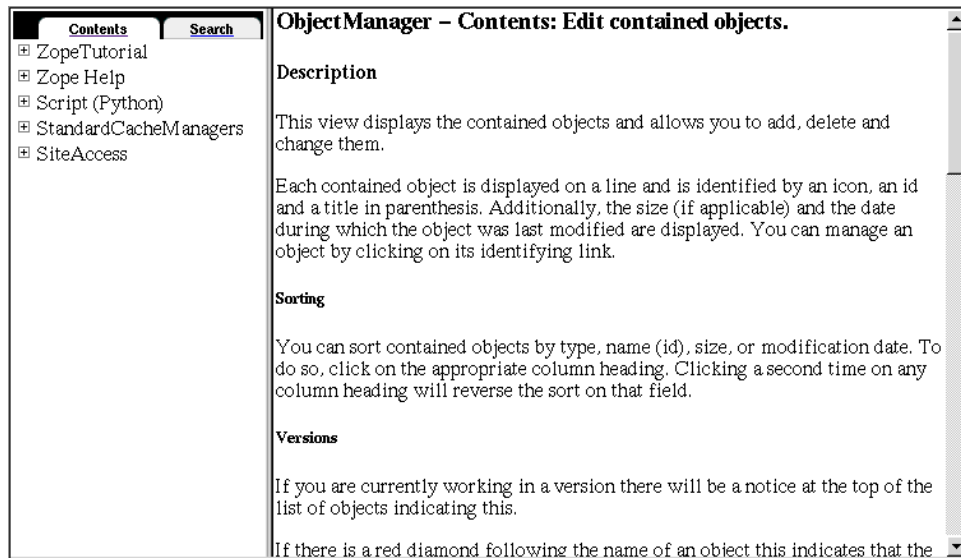


Figure 3-13 The Help System.

The help system has an organization similar 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. You can expand and collapse help topics. To view a help topic in the right frame, click on it in the left frame. By default, no topics are expanded.

Most help pertaining to Zope itself is located in the *Zope Help* folder. Click on the "plus sign" next to the word *Zope Help* in the *Contents* tab of the left frame. The frame will expand to show help topics (in an apparently random and somewhat unhelpful order, currently) and further expandable help categories including *API Reference*, *DTML Reference*, and *ZPT Reference*. These subcategories contain help on scripting Zope, which is explained further in the chapters named *Dynamic Content With DTML*, *Using Zope Page Templates*, and *Advanced Zope Scripting*.

When you install third-party Zope components they may also include help. Each installed component has its own help folder.

You may search for content in the help system by clicking on the Search tab in the left frame, entering one or more search terms. For example, to find all of the help topics that mention folders, type "folder" into the search form and click "Search". This will return a number of help topic links, hopefully most of which pertain to the word "folder".

Logging Out

You may select *Logout* from the Status Frame dropdown box to attempt to log out of Zope. Doing so will cause your browser to "pop up" an authentication dialog. Due to the way most web browsers work, in some cases you actually need to click on the "OK" button with an *incorrect* user name and password filled in to the authentication dialog to really become logged out of the management interface. If you do not do so, you may find that even after selecting "Logout", that you are still logged in. This is an intrinsic limitation of the HTTP Basic Authentication protocol which Zope's stock user folder employs. Alternately, you may close and reopen your browser to log out of Zope.

Using Basic Zope Objects

When building a web application with Zope, you construct the application out of *objects*. The most fundamental Zope objects are explained in this chapter.

Basic Zope Objects

Zope ships with objects that help you perform different tasks. 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, basic Zope objects take on one of 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.

Presentation — You can control the look and feel of your site with Zope objects that act as web page "templates". Zope comes with two facilities to help you manage presentation: DTML (which also handles "logic"), and Zope Page Templates (ZPT). The difference between DTML and ZPT is that DTML allows you to mix presentation and logic, while ZPT does not.

Logic — Zope has facilities for scripting business logic. Zope allows you to script behavior using three facilities: Document Template Markup Language (DTML), Python, and Perl (Perl is only available as an add-on). "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.

The lines between these object categories can become slightly fuzzy. For example, some aspects of DTML fit into *each* of the three categories. But it's mostly for presentation so we stick it in there. Zope also has other kinds of objects that fit into none of these categories. These are explored further in the chapter entitled Zope Services. You may also install "third party" Zope objects to expand Zope's capabilities. These are typically called "Products". You can browse a list of available Zope Products at Zope.org.

Content Objects: Folders, Files, and Images

Folders

You've already met one of the fundamental Zope objects: the *Folder*. Folders are the building blocks of Zope. The purpose of a folder is simple: a Folder's only job in life is *contain* other objects.

Folders can contain any other kind of Zope object, including other folders. 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 Zope security and presentation is 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 using 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.

Files

Zope Files contain raw data, just as the files on your computer do. Software, audio, video and documents are typically transported around the Internet and the world as files. A Zope File object is an analogue to these kinds of files. You can use Files to hold any kind of information that Zope doesn't specifically support, such as Flash files, Java 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 if the content doesn't necessarily need to be edited through the web.

Every File object has a particular *content type* which is a standard Internet MIME designation for types of content. Examples of content types are "text/plain" (plain text content), "text/html" (html text content), and "application/pdf" (an Adobe Portable Document Format file). When you upload a file into Zope, Zope tries to guess the content type from the name of the file.

Creating and Editing Files

To create File object in your Zope instance, visit the root folder and select *File* from Zope's Add list. Before filling out the "id" or "title" of the File object, click the *Browse* button from the resulting "Add File" screen. This should cause your browser to display a dialog box allowing you to choose a "real" file from your local computer which will be uploaded to Zope when the "Add" button on the "Add File" form is selected. Try choosing a file on your local computer such as a Word file (.doc) or a Portable Document Format (.pdf) file.

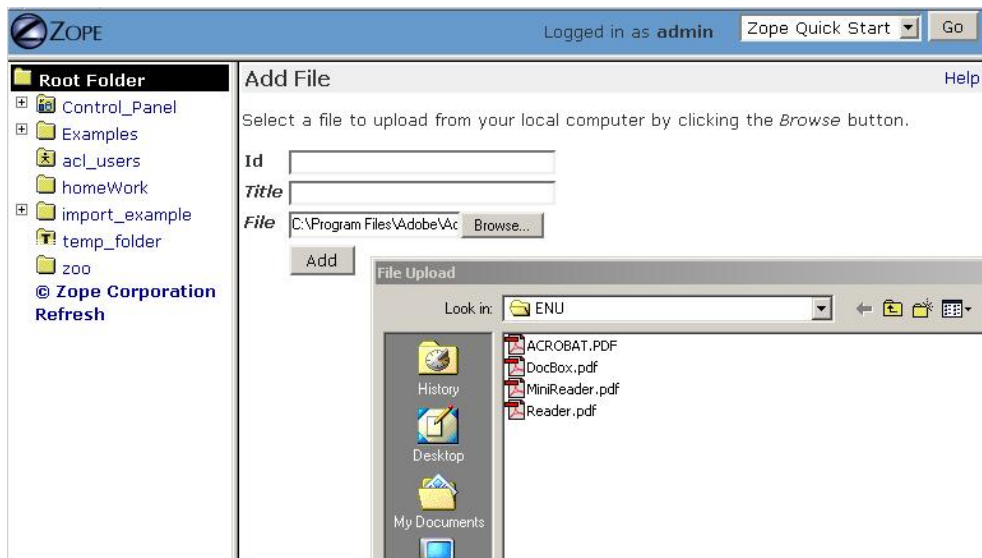


Figure 4-1 Adding a PDF File Object

Zope attempts to use the filename of the file you choose to upload as the File object's `id` and `title`, thus you don't need to supply an `id` or `title` in the "Add File" form unless you want the File object to be named differently than the filename of the file on your local computer. 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 you add the File, a File object with the name of the file on your local computer will appear in the Workspace pane. Look at its *Edit* view. Here you will see that Zope has guessed the content type as shown in the figure below.

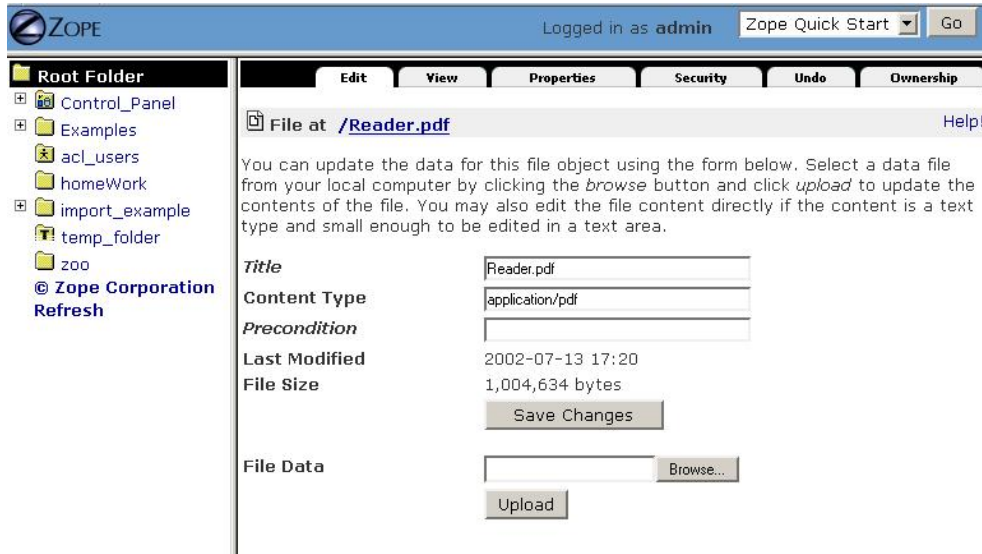


Figure 4-2 Editing an Uploaded PDF File Object

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*. Zope doesn't always guess correctly, so the ability to change the content type of a File object is provided in the editing interface. To change the content type of a File object, type the new content type into the *Content Type* form field and click the *Save Changes* button.

You can specify a *precondition* for a file. A precondition is the name of an executable Zope object (such as a DTML Method, a Script (Python), or an external method) which is executed before the File is viewed or downloaded. If the precondition raises an exception (an error), the file cannot be viewed. This is a seldom-used feature of Files.

You can change the contents of an existing File object by selecting a new file from your local filesystem in the *File Data* form element, clicking *Upload* when the file has been selected.

Editing File Contents

If your File holds only text and is smaller than 64 kilobytes, Zope will allow you to edit its contents in a text area within the Edit view of the management interface. A text file is one that has a content-type that starts with *text/*, such as *text/html*, or *text/plain*.

Viewing Files

You can view a file in the Workspace frame by clicking the *View* tab from a File's management screen.

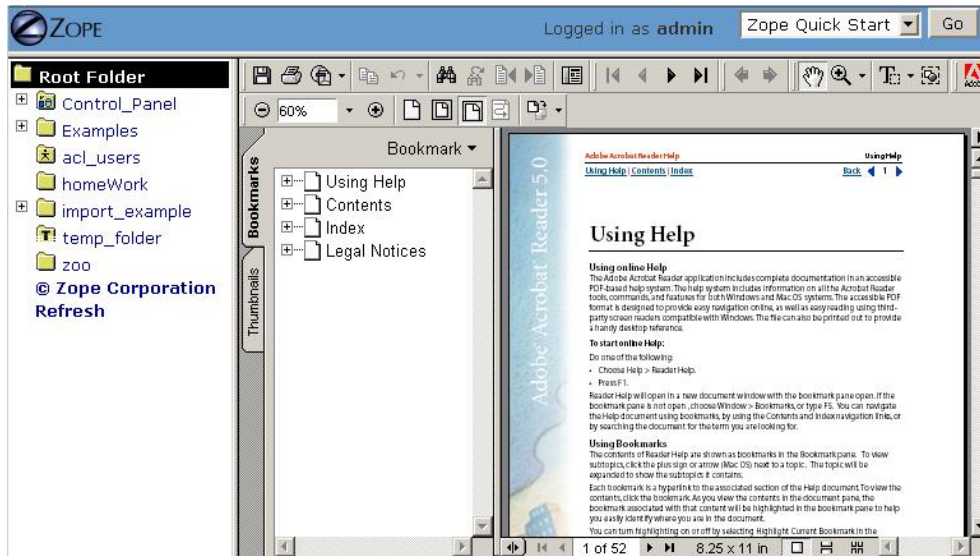


Figure 4-3 Viewing an Uploaded PDF File Object

You can also view a File by visiting its Zope URL. For example, if you have a file in your Zope root folder called *Reader.pdf* then you can view that file in your web browser by going to the URL <http://localhost:8080/Reader.pdf>. Depending on the type of the file and your web browser's configuration, your web browser may choose to display the file or download it.

Images

Image objects contain the data from image files such as GIF, JPEG, and PNG files. In Zope, Images are very 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.

Presentation Objects: Zope Page Templates and DTML Objects

Zope encourages you to keep your presentation and logic separate by providing different objects that are intended to be used expressly for for "presentation". "Presentation" is defined as the task of dynamically defining layout of web pages and other user-visible data. Presentation objects typically render HTML (and sometimes XML or WML).

Zope has two "presentation" facilities: *Zope Page Templates* (ZPT) and *Document Template Markup Language* (DTML). ZPT and DTML are similar to one another but they have slight differences in scope and audience, which are explained in a succeeding section.

Zope Page Templates are objects which allow you to define dynamic presentation for a web page. The HTML in your template is made dynamic by inserting special XML namespace elements into your HTML which define the dynamic behavior for that page.

Document Template Markup Language objects are object which also allow you to define presentation for a web page. The HTML in your template is made dynamic by inserting special "tags" (directives surrounded by angle brackets, typically) into your HTML with define the dynamic behavior for that page.

Both ZPT and DTML are "server-side" scripting languages, like SSI, PHP, embperl, or JSP. This means that DTML and ZPT commands are executed by Zope on 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.

ZPT vs. DTML: Same Purpose, Different Audiences

There is a major problem with many languages designed for the purpose of creating dynamic HTML content: they don't allow for "separation of presentation and logic" very well. For example, "tag-based" scripting languages like DTML, SSI, PHP, and JSP encourage programmers to embed special tags into HTML that are, at best, mysterious to graphics designers who "just want to make the page look good" and don't know (or want to know!) a lot about creating an application around the HTML which they generate. Worse, these tags can sometimes cause the HTML on which the designer has been working to become "invalid" HTML, unrecognizable by any of his or her tools.

Typically when using these kinds of technologies, an HTML designer will "mock up" a page in a tool like Macromedia Dreamweaver or Adobe GoLive. He will then hand it off to a web programmer who will decorate the page with special tags to insert dynamic content. However, using tag-based scripting languages, this is a "one way" function. If the presentation ever needs to change, the programmer cannot just hand back the page that has been "decorated" with the special tags, because these tags will often be ignored or stripped out by the designer's tools. One of several things needs to happen at this point to enact the presentation changes: 1) the designer mocks up a new page and the programmer re-embeds the dynamic tags "from scratch" or 2) the designer hand-edits the HTML, working around the dynamic tags, or 3) the programmer does the presentation himself. Clearly, none of these options are desirable situation, because neither the programmer nor the designer are doing the things that they are best at in the most efficient way.

Zope's original dynamic presentation language was DTML. It soon became apparent that DTML was great at allowing programmers to quickly generate dynamic web pages, but many times failed at allowing programmers to work effectively together with nontechnical graphics designers. Thus, ZPT was born. ZPT is an "attribute-based" presentation language that tries to allow for the "round-tripping" of templates between programmers and nontechnical designers.

Both ZPT and DTML are fully supported in Zope, for now and in the future. Because ZPT and DTML have an overlapping scope, many people are confused about whether to choose one or the other for a given task. A set of "rules of thumb" are appropriate here:

- ZPT is the "tool of choice" if you have a mixed team of programmers and nontechnical designers. Design tools like Macromedia Dreamweaver do not "stomp on" ZPT embedded in a page template, while these tools *do* "stomp on" DTML tags embedded in an HTML page. Additionally, any given ZPT page template is typically viewable in a browser with "default" (static) content even if it has commands embedded in it, which makes it easier for both programmers and designers to preview their work "on the fly". Dynamic DTML content, on the other hand may not be "previewable" in any meaningful way until it is rendered.
- Use DTML when you need to generate non-XML, non-HTML, or non-XHTML-compliant HTML text. ZPT requires that you create pages that are XHTML and/or XML-compliant. ZPT cannot add dynamicism to CSS style sheets, SQL statements, or other non-XML-ish text. DTML excels at it.
- DTML may be easier for some programmers to write because it provides greater control over "conditionals" ("if this, do this, else, do that") than does ZPT. In this respect, it more closely resembles languages such as PHP and ASP-based scripting languages than does ZPT, so it's typically a good "starting place" for programmers coming from these kinds of technologies.

- DTML code can become "logic-heavy" because it does not enforce the "separation of presentation from logic" as strictly as does ZPT. Embedding too much logic in presentation is almost always a bad thing, but is particularly bad when you are working on a "mixed" team of programmers and designers. If you're a "separation of presentation from logic" purist, you will almost certainly prefer ZPT.

Zope Page Templates

Zope Page Templates (ZPTs) are typically used to create dynamic HTML pages.

Creating A Page Template

Create a Folder called *Sales* in the root folder. Enter the *Sales* folder by clicking on it, then select *Page Template* from the Add list. The add form for a page template will be displayed. Specify the *id* "SalesPage" and click *Add*. You have successfully created a page template. Its content is standard "boilerplate" text at this point.

Editing A Page Template

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 textarea where you can edit the template. Click on the "SalesPage" template. You will see something like:

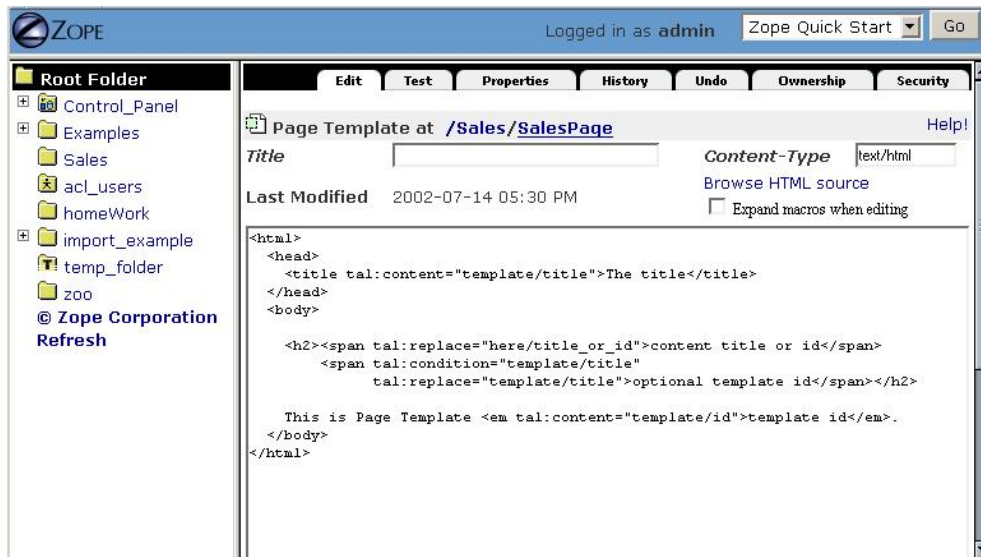


Figure 4-4 Default Page Template Content

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>
```

Then click *Save Changes* at the bottom of the edit form.

Uploading A Page Template

If 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.

Create a text file on your local computer named "test.html". Populate it with the following content:

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

While visiting the Sales folder, select *Page Template* from the add menu, which will cause the page template add form to be displayed. 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 "test.html" file, type in an *Id* of "test" for the new Page Template and click *Add and Edit* . After clicking *Add and Edit* , you will be taken back to the Edit form of your uploaded page template.

Viewing A Page Template

You can view a Page Template in the Workspace frame by clicking the *Test* tab from the template's management screen. Click the *Test* tab of the SalesPage template, and you will see something like the following figure.

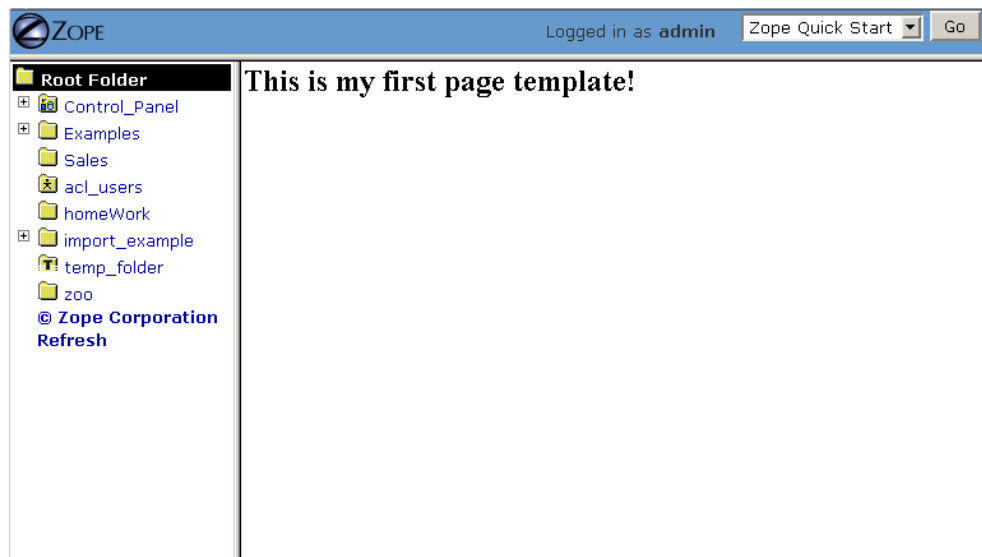


Figure 4-5 Viewing a Page Template

You can also view a Page Template by visiting its Zope URL directly.

DTML Objects: DTML Documents and DTML Methods

DTML is the "other" Zope facility for the creation of presentation in Zope. Two kinds of DTML objects are addable from the Zope Management Interface: *DTML Documents* and *DTML Methods* . Both kinds of objects allow you to perform *security-constrained* presentation logic. The code placed into DTML objects is constrained by Zope's *security policy* , which means, for the most part, that they are unable to import all but a set of restricted Python "modules", and they cannot directly access files on your filesystem. This is actually a "feature", as it allows site administrators to safely delegate the ability to create DTML to "untrusted" or "semi-trusted" users. For more information about Zope's security features, see Users and Security .

A source of frequent confusion for DTML 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 Methods are meant to hold bits of dynamic content that are to be displayed by other DTML Methods and other kinds of Zope objects. For instance, you might create a DTML Method that rendered the content of a navigation bar or a DTML Method that represented a "standard" header for all of your HTML pages. On the other hand, DTML Documents are meant to hold "document-like" content that can stand on its own. DTML Documents also support properties, while DTML Methods do not. The distinction between DTML Methods and DTML Documents is subtle, and if Zope Corporation had it to do "all over again", DTML Documents would likely not exist. (Editor's aside: Believe me, I almost certainly enjoy writing about the difference less than you like reading about it. ;-) There is more information on this topic in the chapters entitled Basic DTML and Variables and Advanced DTML .

As a general rule, you should use a DTML Method to hold DTML content unless you have a really good reason for using a DTML Document, such as a requirement that the container of your DTML content support object properties.

Creating DTML Methods

Click on the Sales folder and then select *DTML Method* from the add list. This process will take you to the add form for a DTML Method. Specify the id "SalesStaff" and the title "The Jungle Sales Staff" and click *Add* . An entry for the new DTML Method object will be displayed in the Contents view of the Workspace pane.

Editing DTML Methods

The easiest and quickest way to edit your newly-created DTML Method is through the management interface. To select your method, click on its name or icon, which will bring up the form shown in the figure below.

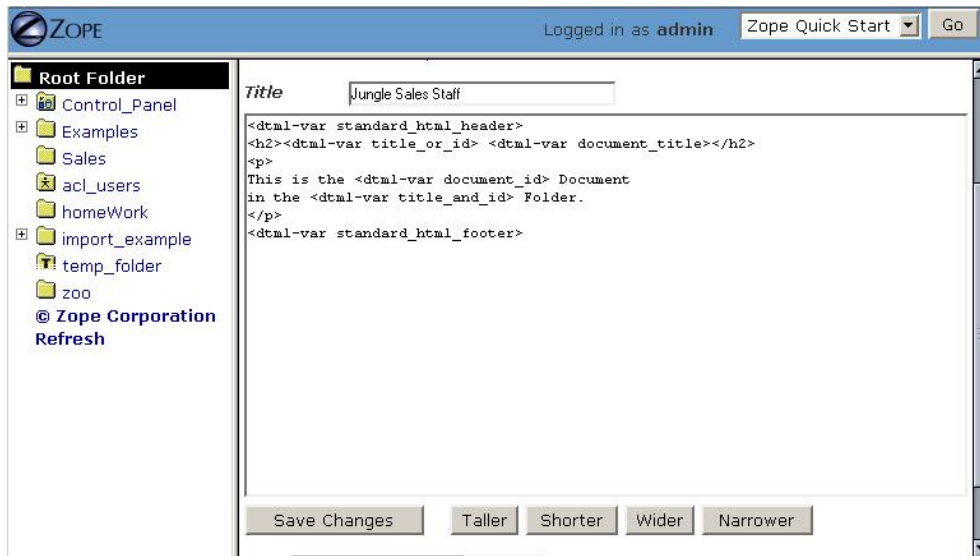


Figure 4-6 Editing a DTML Method

This view shows a text area in which you can edit the content of your document. If you click the *Save Changes* 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 DTML Method. Then add the following HTML content to the textarea:

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

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

Note that the example provided above doesn't do anything "dynamic", it's just some HTML. We will explore the creation of dynamic content with DTML in a later chapter. For now, we're just getting used to using a DTML Method object via the ZMI.

After you have completed the changes to your method, click the *Change* button. Zope returns with a message telling you that your changes have taken effect.

Viewing a DTML Method

You can view a "rendered" DTML Method in the Workspace frame by clicking its *View* tab. Click the *View* tab of the *SalesStaff* DTML method, and you will be presented with something like the following:



Figure 4-7 Viewing a Rendered DTML Method

You can also view a DTML Method by visiting its Zope URL directly.

Uploading an HTML File as Content for a DTML Method

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 Methods.

Create a text file on your local computer named "test.html". Populate it with the following content:

```
<html>
  <body>
    <h1>This is my first uploaded DTML Document!</h1>
  </body>
</html>
```

While visiting the Sales folder, select *DTML Method* from the add menu, which will cause the DTML Method add form to be displayed. 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 "test.html" file, type in an *Id* of "test" for the new DTML Method and click *Add and Edit* . After clicking *Add and Edit* , you will be taken back to the Edit form of your uploaded DTML Method.

Logic Objects: Script (Python) Objects and External Methods

"Logic" objects in Zope are objects which typically perform some sort of "heavy lifting" or "number crunching" in support of presentation objects. When they are executed, they typically do not return HTML or any other sort of structured presentation text. Instead, they typically return values that are easy for a presentation object to format for display. For example, a logic object may return a "list" of "strings". Then, a presentation object may "call in" to the logic object and format the results of the call into a one-column HTML table, where the rows of the table are populated by the strings. Instead of embedding "logic" in a presentation object, you can (and often should) elect to move the logic into a logic object, using a presentation object only to format the result for display. In this manner, you can change or replace the presentation object without needing to "re-code" or replace the logic.

Note that logic objects, like presentation and content objects, are also addressable directly via a URL, and *may* elect to return HTML, which can be displayed in a browser meaningfully. However, the return value of a logic object can almost always be displayed in a browser, even if the logic object does not return HTML.

There are two kinds of logic objects supported by "stock" Zope: *Script (Python)* objects and *External Methods* . An add-on product allows you to code logic in Perl . Several community-contributed Products exist which allow you to use Zope to manage your PHP and JSP scripts, as well, but they are not integrated as tightly as the Python- or Perl-based logic objects. They are PHPParser , PHPObjct , and ZopeJSP .

The "stock" logic objects, External Methods and Script (Python) objects are written in the syntax of the *Python* scripting language. Python is a general-purpose programming language. You are encouraged to read the Python Tutorial in order to understand the syntax and semantics of the example Script (Python) objects and External Methods shown throughout this chapter and throughout this book.

One important Python feature that must be mentioned here, however: Python uses whitespace in the form of indentation to denote block structure. Where other languages, such as C, Perl, and PHP might use "curly braces" to express a block of code, Python determines code blocks by examining the indentation of your code text. If you're used to other programming languages, this may take some "getting-used-to" (typically consisting of a few hours of unsavory spoken language ;-). If you have problems saving or executing Script or External Method objects, make sure to check your Script's indentation.

Script (Python) Objects

Script (Python) objects are one kind of logic object. Note that the torturous form of their name (as opposed to "Python Script") is unfortunate: a legal issue prevents Zope Corporation from naming them "Python Scripts", but most folks at Zope Corporation and in the Zope community refer to them in conversation as just that.

Script (Python) objects are "security-constrained" web-editable pieces of code that are written in a subset of the Python scripting language. Not all Python code is executable via a Script (Python) object. Script (Python) objects are constrained by Zope's *security policy* , which means, for the most part, that they are unable to import all but a set of restricted Python "modules", and they cannot directly access files on your filesystem. This is actually a "feature", as it

allows site administrators to safely delegate the ability to create logic in Python to "untrusted" or "semi-trusted" users. For more information about Zope's security features, see Users and Security .

Creating A Script (Python)

Enter the Sales folder you created earlier by clicking on it, then select *Script (Python)* from the Add list. The add form for the object will be displayed. Specify the `id` "SalesScript" and click *Add* . You will see an entry in the Sales folder Content view representing the "SalesScript" Script (Python) object. Its content is standard "boilerplate" text at this point.

Editing A Script (Python)

The easiest way to edit a Script (Python) 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 Script (Python) which gives you a textarea where you can edit the template. Click on the "SalesScript" icon. You will see something like:

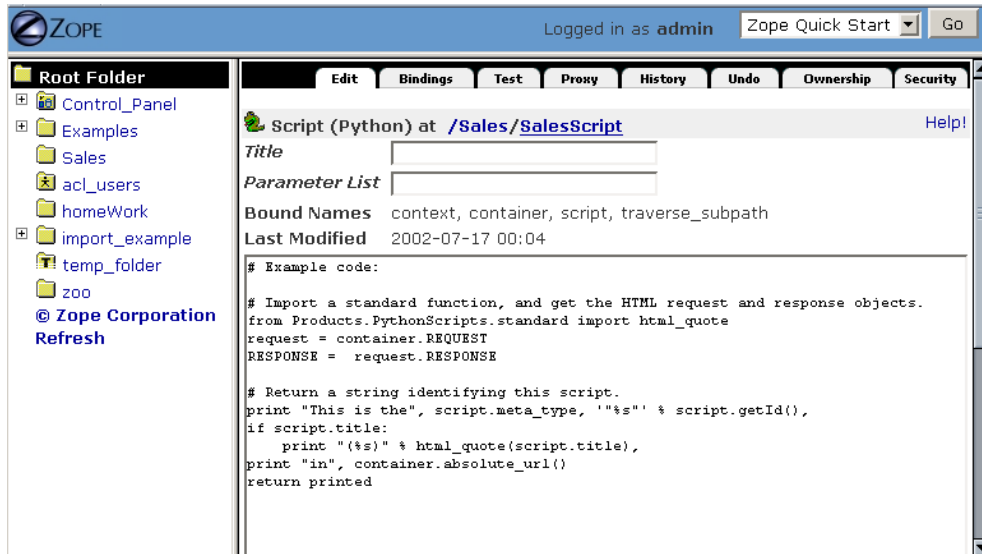


Figure 4-8 Default Script Content

In the *Parameter List* form element, type `name="Chris"` .

Replace the original content that comes in the "body" (the big TEXTAREA) of the Script (Python) object with the following text:

```
return 'Hello, %s from the %s script' % (name, script.id)
```

Then click *Save Changes* at the bottom of the edit form.

Testing A Script (Python)

You can "test" a Script (Python) in the Workspace frame by clicking the *Test* tab from the Script's management screen. When you test a script, the output of the script will be displayed in your browser. Script testing may require that you provide values for the script's *parameters* before you can view the results. Click the *Test* tab of the SalesScript object, and you will see something like the following figure.

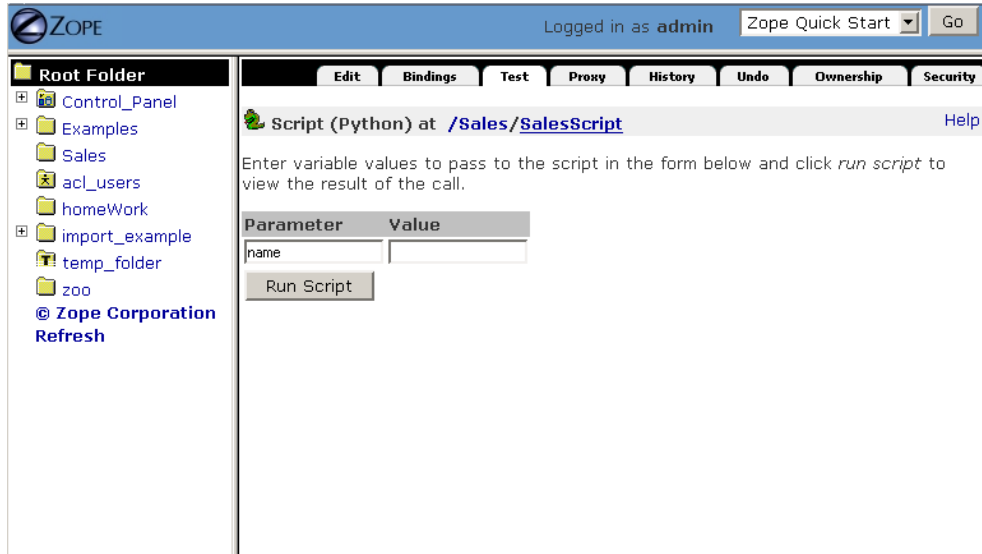


Figure 4-9 Testing a Script

In the Value box next to the `name` parameter, type your name. Then click "Run Script". You will be presented with output in the Workspace frame not unlike:

```
Hello, [yourname] from the SalesScript script
```

If a Script does not require parameters or has defaults for its parameters (as does the example above), you may visit its URL directly to see its output. In our case, visiting the URL of `SalesScript` directly in your browser will produce:

```
Hello, Chris from the SalesScript script
```

If a Script *does* require or accept parameters, you may also influence its execution by visiting its URL directly with a "query string". In our case, visiting the URL `http://localhost:8080/SalesScript?name=Fred` will produce the following output:

```
Hello, Fred from the SalesScript script
```

Zope maps query string argument values to their corresponding parameters automatically, as you can see by this output.

Uploading A Script (Python)

Uploading the body of a Script (Python) is much like uploading the body of a DTML Method or Page Template. One significant difference is that Script (Python) objects interpret text that is offset by "double-pound" (`##`) at the beginning of the text as data about their parameters, title, and "bindings". For example, if you entered the following in a text editor and uploaded it, the lines that start with "double-pound" signs would be interpreted as parameter data, and the only text in the "body" would be the `return` line. It would appear exactly as our `SalesScript` did:

```
## Script (Python) "SalesScript"
##bind container=container
##bind context=context
##bind namespace=
##bind script=script
##bind subpath=traverse_subpath
##parameters=name="Chris"
##title=
##
return 'Hello, %s from the %s script' % (name, script.id)
```

You may see this view of a Script (Python) object by clicking on the `view` or `download` link in the description beneath the "body" textarea.

You may also type the "double-pound" quoted text into the "body" textarea along with the actual script lines and the "double-pound" quoted text will be "automagically" turned into bindings and parameters for the Script.

External Methods

External Methods objects are another kind of logic object. They are very similar to Script (Python) objects. They are scripted in the Python programming language, and they are used for the same purpose. They have a few important differences:

- They are not editable using the Zope Management Interface. Instead, External Methods "modules" need to be created on the filesystem of your Zope server in a special subdirectory of your Zope directory named `Extensions` .
- Because they are not editable via the Zope Management Interface, their execution is not constrained by the Zope "security machinery". This means that unlike Script (Python) objects, they can import and execute essentially arbitrary Python code and access files on your Zope server's file system.
- They do not support the concept of "bindings" (which we have not discussed much, but please just make note for now).

External methods are often useful as an "escape hatch" when Zope's security policy prevents you from using a Script (Python) or DTML to do a particular job that requires more access than is "safe" in through-the-web-editable scripts. For example, a Script (Python) cannot write to files on your server's filesystem, while an External Method may.

Creating and Editing An External Method File

Minimize the browser you're using to access the Zope Management Interface. Open a "shell" console on the machine which you're using as a Zope server. Navigate to the Zope installation folder. You will encounter a subfolder in the Zope installation folder named `Extensions` . Navigate into this folder and create a text file there with the name `SalesEM.py` . Within this file, save the following content:

```
def SalesEM(self, name="Chris"):  
    id = self.id  
    return 'Hello, %s from the %s external method' % (name, id)
```

Creating an External Method Object

Before you can use an External Method from within Zope, you need to create an External Method object in your Zope Management Interface that "refers to" the function in the file that you just created. Reopen your browser window and visit the Zope Management Interface. Navigate to the Sales folder and select *External Method* from the Add list. The Add Form for an External Method will appear. Provide an `Id` of `SalesEM` , a `Title` of `Sales External Method` , a `Module Name` of `SalesEM` and a `Function Name` of `SalesEM` .

Then click *Add* at the bottom of the add form.

Testing An External Method Object

You can "test" an External Method in the Workspace frame by clicking the *Test* tab from the External Method's management screen. When you test an external method, the output of the external method will be displayed in your

browser. Unlike Script (Python) objects, External Methods provide no mechanism for specifying parameter values during testing. However, like Script (Python) objects, their output is influenced by values in a query string when you visit them directly.

Click the *Test* tab of the SalesEM object, and you will see something like the following figure.

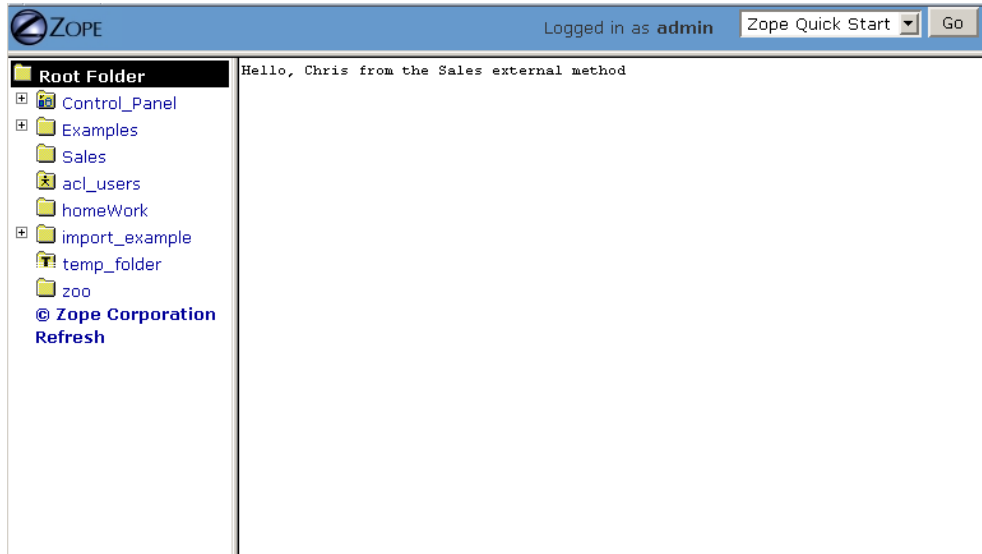


Figure 4-9 Testing an External Method

If an External Method does not require parameters (or has defaults for its parameters, as in the example above), you may visit its URL directly to see its output.

Provide alternate values via a query string to influence the execution of the External Method. For example, visiting the SalesEM external Method via `http://localhost:8080/Sales/SalesEM?name=Fred` will display the following output:

```
Hello, Fred from the Sales external method
```

Alert readers will note that the `id` provided by the output is *not* the `id` of the External Method (`SalesEM`). It is instead the `id` of the "containing" folder, which is named `Sales` ! This is a demonstration of the fact that External Methods (as well as Script (Python) objects) are mostly meant to be used in the "context" of another object, which is often a Folder. This is why they are named methods . Typically, you don't often want to access information about the External Method or Script itself; all the "interesting" information is usually kept in other objects (like Folders). An External Method or Script "knows about" its context and can display information about the context without much fuss.

SQL Methods: Another Kind of Logic Object

SQL Methods are logic objects used to store and execute database queries that you can reuse in your web applications. We don't explain them in this chapter, because we haven't yet explained how to interface Zope with a relational database. SQL Methods are explained in the chapter entitled *Relational Database Connectivity* , where an example of creating a web application using a relational database is given.

Creating a Basic Zope Application Using Page Templates and Scripts

Here is a simple example of using Zope's logic and content objects to build 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 is 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).

We will use Page Template and Script objects to construct an application that will perform this task.

For this example, you will need two Page Templates with the ids *interestRateForm* and *interestRateDisplay* , respectively to collect the information from the user and display it. You will also need a Script (Python) with an id of *calculateCompoundingInterest* that will do the actual calculation.

The first step is to create a folder in which to hold the application. In your Zope's root folder, create a folder named "Interest". You will create all of the objects which follow within this folder.

Creating a Data Collection Form

Visit the *Interest* folder by clicking on it within the Zope Management Interface. Within the *Interest* folder, create a Page Template with the id *interestRateForm* that collects "principal", "interest_rate", "periods" and "years" from your users. Use this text as the body of your *interestRateForm* page template:

```
<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, when it is submitted, calls the *interestRateDisplay* template (which we have not yet created).

Creating A Script To Calculate Interest Rates

Now, revisit the Contents view of the *Interest* folder and create a Script (Python) object with the id *calculateCompoundingInterest* that accepts four parameters: *principal* , *interest_rate* , *periods* and *years* . Provide it with the following "body":

```
"""
Calculate compounding interest.
```

```
"""
i = interest_rate / periods
n = periods * years
return ((1 + i) ** n) * principal
```

Remember: you enter the parameter names, separated by commas, into the *Parameters List* field, and the body into the body text area. Remember also that when you're creating a Script (Python) object, you're actually programming in the Python programming language which is indentation-sensitive. Make sure each of the lines above line up along the very left side of the text area, or you may get an error when you attempt to save it.

Creating A Page Template To Display Results

Next, go back to the Contents view of the *Interest* folder and create a Page Template with the id *interestRateDisplay*. This Page Template is **called by** *interestRateForm* and **calls** *calculateCompoundingInterest*. It also renders and returns the results:

```
<html>
  <body>
    Your total balance (or debt) including compounded interest over
    <span tal:define="years request/years;
                    principal request/principal;
                    interest_rate request/interest_rate;
                    periods request/periods">
    <span tal:content="years">2</span> years is:<br><br>
    <b>§
    <span tal:content="python: here.calculateCompoundingInterest(principal,
                                                                interest_rate,
                                                                periods,
                                                                years)" >1.00</span>

    </b>
  </span>
</body>
</html>
```

Dealing With Errors

In any programming venue, you will need to deal with errors. Nobody's perfect! You may have already encountered some as you've entered these scripts. Let's explore errors a bit by way of an example. In our case, we cannot use the Page Template *Test* tab to test the *interestRateDisplay* without receiving an error, because it depends on the *interestRateForm* to supply it with the variables "years", "principal", "interest_rate" and "periods". It is not directly "testable". For the sake of "seeing the problem before it happens for real", click the *Test* tab. Zope will present an error page with text not unlike the text below:

```
Zope Error

Zope has encountered an error while publishing this resource.

Error Type: KeyError
Error Value: years
```

This error message is telling you that your Page Template makes a reference to a variable "years" that it can't find. If you've created a *Site Error Log* object in your root folder (it will be named *error_log*), you can view the full error by visiting the *error_log* object and clicking the topmost error log entry link which will be name *KeyError: years* on the *Log* tab. The error log entry will be displayed. It contains information about the error, including the time, the user who received the error, the URL which caused the error to happen, the exception type, the exception value, and a "Traceback" which typically gives you enough information to understand what happened. In our case, the part of the traceback that is interesting to us is:

```
* Module Products.PageTemplates.TALES, line 217, in evaluate
URL: /Interest/interestRateDisplay
Line 4, Column 8
```

```
Expression: standard:'request/years'
```

This tells us that the failure occurred when the Page Template attempted to access the variable `request/years`. We know why: there is no variable `request/years`, because that variable is only "filled in" as a result of posting via our `interestRateForm`, which calls in to our `interestRateDisplay` Page Template, which has the effect of inserting the variables `principal`, `interest_rate`, `periods` and `years` into `request` "namespace". We'll cover Page Template namespaces in a succeeding chapter, but for now, let's move on.

Using The Application

Let's use the application you've just created. Visit the `interestRateForm` Page Template and click the `Test` tab.

Type in 20000 for balance or debt, .06 for interest rate, 4 for periods in a year, and 20 for number of years and click `Calculate`. This will cause `interestRateForm` to submit the collect information to `interestRateDisplay`, which calls the Script (Python) named `calculateCompoundingInterest`. The display method uses the value returned by the script in the resulting display. You will see the following result.

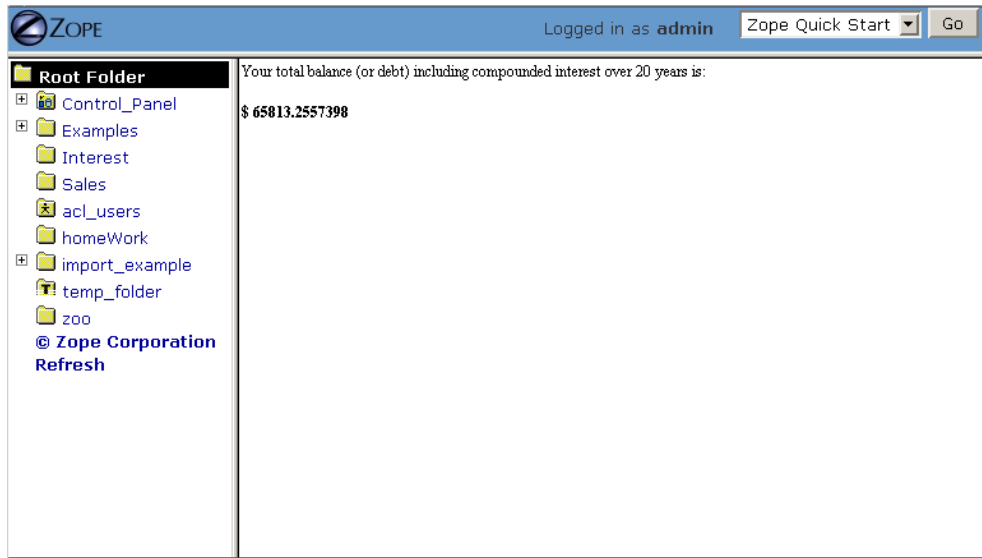


Figure 4-10 Result of the Interest Application

If you see something close to this, it calls for congratulations, because you've just built your first Zope application successfully. If you are having troubles, try to troubleshoot the application by using the tips in the section above "Dealing With Errors." If you're stuck entirely, it's advisable that you send a message to the Zope mailing list detailing the problem that you're having in as concise and clear a form as possible. It is likely that someone there will be able to help you. It is polite to subscribe to the maillist itself if you want to receive replies. See the Mailing list section of Zope.org for information about how to subscribe to the Zope (`zope@zope.org`) maillist.

The Zope Tutorial

Zope comes with a built-in tutorial which reinforces some of the concepts you've learned here. As an extension of this book, we recommend that you run the tutorial to get a feel for using basic Zope objects (particularly DTML objects). To use the tutorial properly, your browser should support *JavaScript* and *cookies*.

To launch the tutorial, navigate to the root folder and add a Zope Tutorial object by selecting *Zope Tutorial* from the add list. When the add form asks for an "id" for the object, give it the id *tutorial* and click "Add". You will be directed to a

screen with a "Begin Tutorial" button. When you click the "Begin Tutorial" button, a new browser window resembling the help system will be opened with the tutorial. If another window does not appear, either your browser does not support *JavaScript* or it is configured to disallow the opening of new windows. This will prevent you from being able to use the tutorial, so you may want to try a different browser.

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 visiting the help system and navigating to the *Zope Tutorial* help category. There is no need to re-install the tutorial.

Acquisition

Acquisition is the technology that allows dynamic behavior to be shared between Zope objects via *containment* .

Acquisition's flavor permeates Zope. It can be used almost everywhere within Zope: in DTML, in Zope Page Templates, in Script (Python) objects, and even in Zope URLs. Because of its ubiquity in Zope, a basic understanding of acquisition is important.

Acquisition vs. Inheritance

The chapter entitled Object Orientation describes a concept called *inheritance* . Using inheritance, an object can *inherit* some of the behaviors of a specific class, *overriding* or adding other behaviors as necessary. Behaviors of a class are nearly always defined by its *methods* , although attributes can be inherited as well.

In a typical object-oriented language, there are rules to the way a *subclass* inherits behavior from its *superclasses* . For example, in Python (a *multiple-inheritance* language), a class may have more than one superclass, and rules are used to determine which of a class' superclasses is used to define behavior in any given circumstance. We'll define a few Python classes here to demonstrate. You don't really need to know Python inside and out to understand these examples. Just know that a `class` statement defines a class and a `def` statement inside of a class statement defines a method. A class statement followed by one or more words inside (Parenthesis) causes that class to *inherit* behavior from the classes named in the parenthesis.:

```
class SuperA:
    def amethod(self):
        print "I am the 'amethod' method of the SuperA class"

    def anothermethod(self):
        print "I am the 'anothermethod' method of the SuperA class"

class SuperB:
    def amethod(self):
        print "I am the 'amethod' method of the SuperB class"

    def anothermethod(self):
        print "I am the 'anothermethod' method of the SuperB class"

    def athirdmethod(self):
        print "I am the 'anothermethod' method of the SuperB class"

class Sub(SuperA, SuperB):
    def amethod(self):
        print "I am the 'amethod' method of the Sub class"
```

If we make an *instance* of the "Sub" class, and attempt to *call* one of its methods, there are rules in place to determine whether the behavior of the method will be defined by the Sub class itself, its SuperA superclass, or its SuperB superclass. The rules are fairly simple. If the Sub class has itself defined the named method, that method definition will be used. Otherwise, the *inheritance hierarchy* will be searched for a method definition.

The *inheritance hierarchy* is defined by the class' superclass definitions. In the case of the Sub class above, it has a simple inheritance hierarchy: it inherits first from the SuperA superclass, then it inherits from the SuperB superclass. This means that if you call a method on an instance of the Sub class, and that method is not defined as part of the Sub class' definition, it will first search for the method in the SuperA class and if it doesn't find it there, it will search in the SuperB class.

Here is an example of calling methods on an instance of the above-defined Sub class using the Python interpreter:

```
>>> instance = Sub()
```

```
>>> instance.amedithod()
I am the 'amedithod' method of the Sub class
>>> instance.anothermethod()
I am the 'anothermethod' method of the SuperA class
>>> instance.athirdmethod()
I am the 'anothermethod' method of the SuperB class
```

Note that when we called the `anothermethod` method on the Sub instance, we got the return value of SuperA's method definition for that method, even though both SuperA and SuperB defined that method. This is because the inheritance hierarchy specifies that the first superclass (SuperA) is searched first.

The point of this example is that instances of objects use their *inheritance hierarchy* to determine their behavior. In non-Zope applications, this is the only way that object instances know about their set of behaviors. However, in Zope, objects make use of another facility to search for their behaviors: *acquisition* .

Acquisition is about Containment

The concept behind acquisition is simple:

- Objects are situated inside other objects. These objects act as their "containers". For example, the container of a DTML Method named "amedithod" inside the DTML_Example folder is the DTML_Example folder.
- Objects may acquire behavior from their containers.

Inheritance stipulates that an object can learn about its behavior from its superclasses via an *inheritance hierarchy* . *Acquisition* , on the other hand, stipulates that an object can additionally learn about its behavior its through its *containment hierarchy* . In Zope, an object's inheritance hierarchy is always searched for behavior before its acquisition hierarchy. If the method or attribute is not found in the object's inheritance hierarchy, the acquisition hierarchy is searched.

Say What?

Let's toss aside the formal explanations. Acquisition can be best explained with a simple example.

Place a DTML Method named `acquisition_test` in your Zope root folder. Give it the following body:

```
<html>
<body>
  <p>
    I am being called from within the <dtml-var id> Folder!
  </p>
</body>
</html>
```

Save it and then use the DTML Method "View" tab to see the result of the DTML method in your Workspace frame. You will see something not unlike the following:

```
I am being called from within the Zope Folder!
```

The `id` of the Zope root folder is `Zope` , so this makes sense. Now create a Folder inside your Zope root folder named `AcquisitionTestFolder` . We're going to invoke the `acquisition_test` method *in the context of the* `AcquisitionTestFolder` folder. To do this, assuming your Zope is running on your local machine on port 8080, visit the URL `http://localhost:8080/AcquisitionTestFolder/acquisition_test` . You will see something not unlike the following:

```
I am being called from within the AcquisitionTestFolder Folder!
```

Note that even though an object named `acquisition_test` does not "live" inside the `AcquisitionTestFolder` folder, Zope found the method and displayed a result anyway! Not only did Zope display a result, instead of inserting the `id` of the Zope root folder, it inserted the `id` of the `AcquisitionTestFolder` folder! This is an example of acquisition in action. The concept is simple: if a named object is not found as an attribute of the object you're searching, its containers are searched until the object is found. In this way, acquisition can *add behavior* to objects. In this case, we added a behavior to the `AcquisitionTestFolder` folder that it didn't have before (by way of giving it an `acquisition_test` method).

Providing Services

It can be said that acquisition allows objects to acquire *services* by way of containment. For example, our `AcquisitionTestFolder` folder acquired the services of the `acquisition_test` method.

Not only do objects acquire services, they also provide them. For example, adding a `Mail Host` object to a `Folder` named `AFolder` provides other objects in that folder with the ability to send mail. But it also provides objects contained in *subfolders* of that folder with the capability to send mail. If you create subfolders of `AFolder` named `AnotherFolder` and `AThirdFolder`, you can be assured that objects placed in *these* folders will also be able to send mail in exactly the same way as objects placed in `AFolder`.

Acquisition "goes both ways". When you create an object in Zope, it has the capability to automatically acquire services. Additionally, it automatically provides services that other objects can acquire. This makes reuse of services very easy since you don't have to do anything special to make services available to other objects.

Getting Deeper with Multiple Levels

If you place a method in the root folder, and create a subfolder in the root folder, you can acquire the method's behaviors. So what happens if things get more complex? Perhaps you have a method that needs to be acquired from within a couple of folders. Is it acquired from its parent, or its parent's parent, or what?

The answer is that acquisition works on the entire object hierarchy. If for example you have a `DTML Method` in the root folder. Also in the root folder you have three nested `Folders` named "Users", "Barney" and "Songs". You may call this URL:

```
/Users/Barney/Songs/HappySong
```

The `HappySong` method is found in the root folder unless one of the other folders "Users", "Barney" or "Songs" happens to also have a method named "HappySong", in which case *that* method is used. The `HappySong` method is searched for first directly in the "Songs" folder. If it is not found, the acquisition hierarchy is searched starting at the first container in the hierarchy: "Barney". If it is not found in "Barney", the "Users" folder is searched. If it is not found in the "Users" folder, the root folder is searched. This search is called *searching the acquisition path* or alternately *searching the containment hierarchy*.

Acquisition is not limited to searching a containment hierarchy: it can also search a *context hierarchy*. Acquisition by context is terribly difficult to explain, and you should avoid it if possible. However, if you want more information about acquiring via a context and you are ready to have your brain explode, please see the presentation named `Acquisition Algebra`.

Summary

Acquisition allows behavior to be distributed throughout the system. When you add a new object to Zope, you don't need to specify all its behavior, only the part of its behavior that is unique to it. For the rest of its behavior it relies on other objects. This means that you can change an object's behavior by changing where it is located in the object

hierarchy. This is a very powerful function which gives your Zope applications flexibility.

Acquisition is useful for providing objects with behavior that doesn't need to be specified by their own methods or methods found in their inheritance hierarchies. Acquisition is particularly useful for sharing information (such as headers and footers) between objects in different folders as well. You will see how you can make use of acquisition within different Zope technologies in upcoming chapters.

A more exhaustive technical explanation of the underpinnings of Zope's acquisition technology is available in the Zope Developer's Guide .

Basic DTML

DTML (Document Template Markup Language) is a templating facility which supports the creation of dynamic HTML and text. It is typically used in Zope to create dynamic web pages. For example, you might use DTML to create a web page which "fills in" rows and cells of an HTML table contained within the page from data fetched out of a database.

DTML is a *tag-based* presentation and scripting language. This means that *tags* (e.g. `<dtml-var name>`) embedded in your HTML cause parts of your page to be replaced with "computed" content.

DTML is a "server-side" scripting language. 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.

How DTML Relates to Similar Languages and Templating Facilities

DTML is similar in function to "HTML-embedded" scripting languages such as JSP, PHP, or `mod_perl`. It differs from these facilities inasmuch as it will not allow you to create "inline" Python *statements* (if... then.. else..) in the way that JSP, `mod_perl` or PHP will allow you to embed a block of their respective language's code into an HTML page. DTML does allow you to embed Python *expressions* (`a == 1`) into HTML-like tags. It provides flow control and conditional logic by way of "special" HTML tags. It is more similar to Perl's `HTML::Template` package than it is to `mod_perl` in this way. It can also be compared to the web server facility of Server Side Includes (SSI), but with far more features and flexibility.

When To Use DTML

If you want to make a set of dynamic web pages that share bits of content with each other, and you aren't working on a project that calls for a tremendous amount of collaboration between programmers and tool-wielding designers, DTML works well. Likewise, if you want to dynamically create non-HTML text (like CSS stylesheets or email messages), DTML can help.

When Not To Use DTML

If you want code which expresses a set of complex algorithms to be maintainable (as "logic" programming should be), you shouldn't write it in DTML. DTML is not a general purpose programming language, it instead is a special language designed for formatting and displaying content. While it may be possible to implement complex algorithms in DTML, it is often painful.

For example, let's suppose you want to write a web page which displays a representation of the famous Fibonacci sequence . You would not want to write the program that actually makes the calculation of the Fibonacci numbers by writing DTML. It could be done in DTML, but the result would be difficult to understand and maintain. However, DTML is perfect for describing the page that the results of the Fibonacci calculations are inserted into. You can "call out" from DTML to Script (Python) objects as necessary and process the results of the call in DTML. For example, it is trivial in Python (search for the word Fibonacci on this page) to implement a Fibonacci sequence generator, and trivial in DTML to create a dynamic web page which shows these numbers in a readable format. If you find yourself creating complex and hard-to-understand logic in DTML, it's likely time to explore the the Zope features which allow you to script "logic" in Python, while letting DTML do the presentation "dirty work".

String processing is another area where DTML is likely 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, which has more powerful string processing capabilities than DTML.

Zope has a technology named Zope Presentation Templates which has purpose similar to DTML. DTML and ZPT are both facilities which allow you to create dynamic HTML. However, DTML is capable of creating dynamic text which is *not* HTML, while ZPT is limited to creating text which is HTML (or XML). DTML also allows users to embed more extensive "logic" in the form of conditionals and flow-control than does ZPT. While the source to a ZPT page is almost always "well-formed" HTML through its lifetime, the source to DTML pages are not guaranteed to be "well-formed" HTML, and thus don't play well in many cases with external editing tools such as Dreamweaver.

Both ZPT and DTML are fully supported technologies in Zope, and neither is "going away" any time soon. A discussion about when to use one instead of the other is available in the chapter entitled Using Basic Zope Objects in the section entitled "ZPT vs. DTML: Same Purpose, Different Audiences", but the choice is sometimes subjective.

The Difference Between DTML Documents and DTML Methods

You can use DTML scripting commands in two types of Zope objects, *DTML Documents* and *DTML Methods*. These two types of DTML objects are subtly different from one another, and their differences cause many would-be DTML programmers to become confused when deciding to use one versus the other. So what is the difference?

DTML Methods are used to carry out actions. They are *presentation* objects (as used in the vernacular of the Using Basic Zope Objects chapter). If you want to render the properties or attributes of another object like a DTML Document or a Folder, you will use a DTML Method. DTML Methods do not have their own properties.

DTML Documents are *content* objects (in the vernacular used in the chapter entitled Using Basic Zope Objects). If you want to create a "stand-alone" HTML or text document, you might create a DTML Document object to hold the HTML or text. DTML Document objects have their own *properties* (attributes), unlike DTML Methods.

In almost all cases, you will want to use a DTML Method object to perform DTML scripting. DTML Document objects are an artifact of Zope's history that is somewhat unfortunate. In Zope's earlier days, a consensus came about that it was important to have objects in Zope that could perform DTML commands but have properties of their own. At the time, the other content objects in Zope, such as Files and Images were either nonexistent or had limitations in functionality that made the concept of a DTML Document attractive. That attraction has waned as Zope's other built-in content objects have become more functional. DTML Documents remain in Zope almost solely as a backwards-compatibility measure. If you never use a DTML Document in your work with Zope, you won't miss out on much!

Details

DTML Methods are method objects. The chapter named Object Orientation discusses the concept of a "method". DTML Methods are *methods* of the folder that contains them, and thus they do not have regard for their own identity as a Zope object when they are used. For example, if you had a folder called Folder and a DTML method in that folder called Method:

```
AFolder/  
    AMethod
```

AMethod is a *method* of AFolder. This means that AMethod does not have any of it's own attributes or properties. Instead it uses those of AFolder. Suppose you put the following DTML string in AMethod:

```
<dtml-var id>
```

When you view the AMethod DTML Method, you will see the string `AFolder`, which is the `id` of AMethod's containing Folder (AFolder). When this DTML method is viewed, it resolves the name `id` to the string which is the value of AFolder's `id` property.

DTML Documents, on the other hand, are not methods. They are "aware" of their own identity as Zope objects. For example, if you created a DTML Document in the folder AFolder called ADocument, and you put the above DTML string into ADocument and viewed it, it would render to the string ADocument . It resolves the name id to the string which is the value of its own id, not the id of its containing folder.

For this chapter, unless stated otherwise, use DTML Methods to hold the example DTML text, as opposed to DTML Documents!

DTML Tag Syntax

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 with a `</dtml-var>` tag because it is a singleton 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.

DTML Tag Names, Targets, and Attributes

All DTML tags have *names* . The name is simply the word which follows dtml- . For instance, the name of the DTML tag dtml-var is var , and the name of the DTML tag dtml-in is in .

Most DTML tags have *targets* . The target of a DTML tag is just the word or expression that, after a space, follows the tag name. For example, the target of the DTML tag `<dtml-var standard_html_header>` is `standard_html_header` . The target of the DTML tag `<dtml-in foo>` is `foo` . The target of the DTML tag `<dtml-var "objectIds()">` is the expression "objectIds()". The target typically refers to the name of an object (or a Python expression that resolves to an object) that you wish the tag to operate upon.

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 its target. 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>
```

Here we are referencing the *exclamation* target, modifying it with the attribute *upper* . Notice that the *upper* attribute, unlike the *missing* attribute doesn't need a value.

See the DTML Reference appendix for more information on the syntax of different DTML tags.

Creating a "Sandbox" for the Examples in This Chapter

You should create a Folder in your Zope's root folder named "DTML_Examples" if you intend on creating objects from examples in this chapter. Create the example objects within this "sandbox". This prevents you from littering your Zope root folder with DTML examples.

Examples of Using DTML for Common Tasks

Below, we show how to use DTML to complete three common tasks: inserting text into a web page, displaying results by iterating over a sequence, and processing form results.

Inserting Text into HTML with DTML

DTML commands are written as tags that begin with *dtml-*. You create dynamic content in DTML by mixing HTML tags and DTML tags together. Inserting the value of a variable (a variable is also known as a "target") into HTML is the most basic task that you can perform with DTML. Many DTML tags insert variable values, and they all do it in a similar way. Let's look more closely at how Zope inserts variable values.

Create a folder in your sandbox with the id "Feedbags" and the title "Bob's Fancy Feedbags". While inside the Feedbags 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. When you view the DTML method this way, it will be *rendered*, which means that you will not necessarily see a straight representation of the HTML that you typed in to the form. Instead you will see the *rendered* version of the page, which will include the extra text provided by DTML by way of the tags you've inserted. You should see something like the figure below::

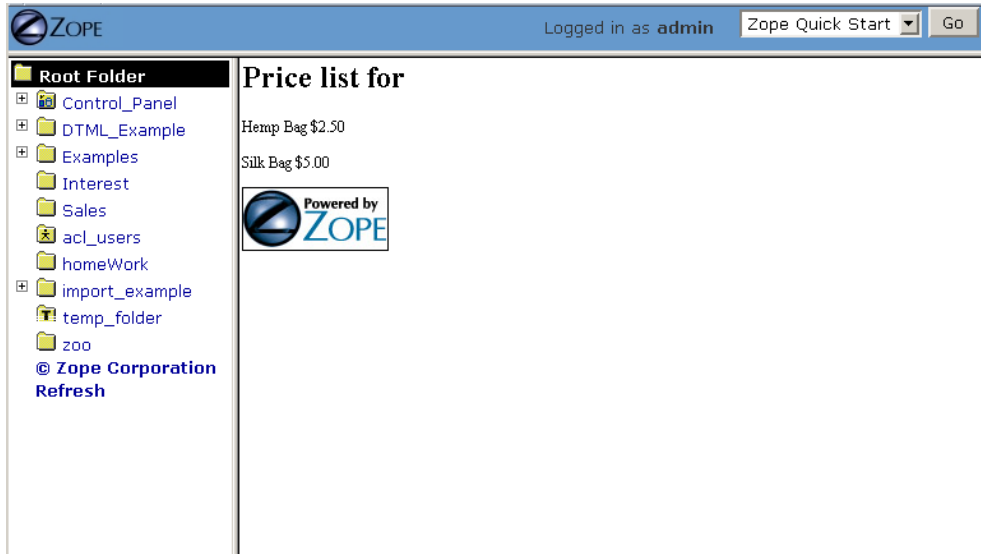


Figure 9-1 Viewing the pricelist method

If you tell your browser to view the HTML source of the Workspace frame, you will see something not unlike the below:

```
<html><head><title>Feedbags</title></head><body bgcolor="#FFFFFF">
<h1>Price list for </h1>
<p>Hemp Bag $2.50</p>
<p>Silk Bag $5.00</p>
<p><a href="http://www.zope.org/Credits" target="_top">
  
</a>
</p>
</body>
</html>
```

That's certainly not what you typed in, is it?

DTML makes the reuse of content and layout possible. In the example above, we've made use of the `standard_html_header` DTML Method and the `standard_html_footer` DTML Method, both of which live in the root folder, to insert HTML text into our page. These DTML methods (and any other DTML method) can be used by other DTML methods to insert text into our rendered output.

We've seen that DTML inserts an HTML header, an HTML footer, and a title into the web page. But how does the "var" tag *find* the values that it inserts in place of "standard_html_header", "title" and "standard_html_footer"?

DTML name lookup is somewhat "magical", because you don't need to explicitly tell DTML *where* to find a variable. Instead, it tries to guess what you mean by following a preordained set of search rules. DTML gets the values for variable names by searching an environment which includes the current object, the containment path, and request variables like values submitted by a form and cookies. The DTML Name Lookup Rules represent the namespaces searched and their relative precedence. As an example, let's follow the `pricelist` DTML code step-by-step. In our `pricelist` method, we've asked DTML to look up three names: "standard_html_header", "title", and "standard_html_footer". It searches for these variables in the order that they are mentioned in the page.

DTML looks first for "standard_html_header". It looks in the "current object" first, which is its container, the `Feedbags` folder. The `Feedbags` folder doesn't have any methods or properties or sub-objects by that name. Next Zope tries to

acquire the object from its containers. It examines the `Feedbags` folder's container (your sandbox folder, likely named "DTML_Examples"), which also doesn't turn up anything. It continues searching through any intermediate containers, which also don't have a method or property named "standard_html_header" unless you've put one there. It keeps going until it gets to the root folder. The root folder *does* have a sub-object named "standard_html_header", which comes as a default object in every Zope. The `standard_html_header` object is a DTML Method. So Zope *calls* the `standard_html_header` method in the root folder and inserts the results into the page. Note that once DTML *finds* a property or variable, if it is callable (as in the case of a DTML Method, an External Method, a SQL Method, or a Script (Python) object), it is called and the results of the call are inserted into the page.

Next DTML looks for the name "title". Here, the search is a shorter. On its first try, DTML finds the `Feedbags` folder's `title` property and inserts it. The `title` property is not a method or a script, so DTML doesn't need to *call* it. It just renders it into the output.

Finally DTML looks for the name `standard_html_footer`. It has to search all the way up to the root folder to find it, just like it looked for `standard_html_header`. It calls the `standard_html_footer` in the root and inserts the text result.

The resulting page is fully assembled (rendered) at this point, and is sent to your browser.

Understanding how DTML looks up variables is important. We will explore the DTML name lookup mechanism further in the chapter entitled Advanced DTML. It is also documented in Appendix E.

Formatting and Displaying Sequences

It is common that people want to use DTML to format and display *sequences*. A sequence is just a list of items, like "Fred, Joe, Jim". Often, you want to create an HTML table or a bulleted list that contains elements in a sequence. Let's use DTML to call out to an object which returns a sequence and render its result.

Create a Script (Python) object in your sandbox folder named "actors". Give the script the following body and save it:

```
## Script (Python) "actors"
##bind container=container
##bind context=context
##bind namespace=
##bind script=script
##bind subpath=traverse_subpath
##parameters=
##title=
##
return ['Jack Lemmon', 'Ed Harris', 'Al Pacino', 'Kevin Spacey', 'Alan Arkin']
```

Make sure that all of the lines of this script line up along the left-hand side of the textarea to avoid receiving an error when you attempt to save the script, since Python is sensitive to indentation. This Script (Python) object returns a Python data structure which is a *list of strings*. A list is a kind of *sequence*, which means that DTML can *iterate* over it using the *dtml-in* tag. Now create a DTML Method named "showActors" in your sandbox, give it this body, and save it:

```
<html>
<body>
<h1>Actors in the movie Glengarry Glen Ross</h1>
<table border="1">
  <th>Name</th>
<dtml-in actors>
  <tr>
    <td><dtml-var sequence-item></td>
  </tr>
</dtml-in>
</table>
</body>
</html>
```

The DTML *in* tag iterates over the results of the *actors* script and inserts a table row into a table for each of the actors mentioned in the script. Note that inside the table cell, we use a special name *sequence-item*. *sequence-item* is a special name that is meaningful within a *dtml-in* tag. It refers to the "current item" (in this case, the actor name string) during processing. The HTML source of the Workspace frame when you click the *View* tab on the `showActors` method will look something like:

```
<html>
<body>
<h1>Actors in the movie Glengarry Glen Ross</h1>
<table border="1">
  <th>Name</th>
  <tr>
    <td>Jack Lemmon</td>

  </tr>
  <tr>
    <td>Ed Harris</td>
  </tr>
  <tr>
    <td>Al Pacino</td>
  </tr>
  <tr>
    <td>Kevin Spacey</td>
  </tr>
  <tr>
    <td>Alan Arkin</td>
  </tr>
</table>
</body>
</html>
```

Note that you didn't have to specifically tell DTML that you are querying a Script (Python) object. You just tell it the name of the object to call (in this case `actors`), and it does the work of figuring out how to call the object and pass it appropriate arguments. If you replace the `actors` Script with some other kind of object that does exactly the same thing, like another DTML Method, you won't have to change your `showActors` DTML Method. It will "just work".

Processing Input from Forms

You can use DTML to perform actions based on the information contained in the submission of an HTML form.

Create a DTML Method named "infoForm" with the following body:

```
<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_addr"><br>
<input type="submit" name="submit" value=" Submit " >
</form>

<dtml-var standard_html_footer>
```

This is a web form that asks the user for information, specifically his user name and email address. Note that you refer to the name "infoAction" as the *action* of the HTML form. This really has nothing to do with DTML, it's an attribute of the HTML *form* tag. But the name specified in the form action tag can name another Zope object which will receive and process the results of the form when it is submitted.

Create a DTML Method named *infoAction* in the same folder as the `infoForm` method. This is the *target* of the `infoForm` form action. This method will display a bland "thanks" message which includes the name and email

information that was gathered from the web form. Provide the `infoAction` method with the following body and save it:

```
<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_addr> describing our aardvark adoption
program as soon as it receives final governmental approval.
</p>

<dtml-var standard_html_footer>
```

Navigate back to the `infoForm` method and use the *View* tab to execute it. Fill out the form and click the *Submit* button. If all goes well you should see a thank you message that includes your name and email address, much like the figure below::

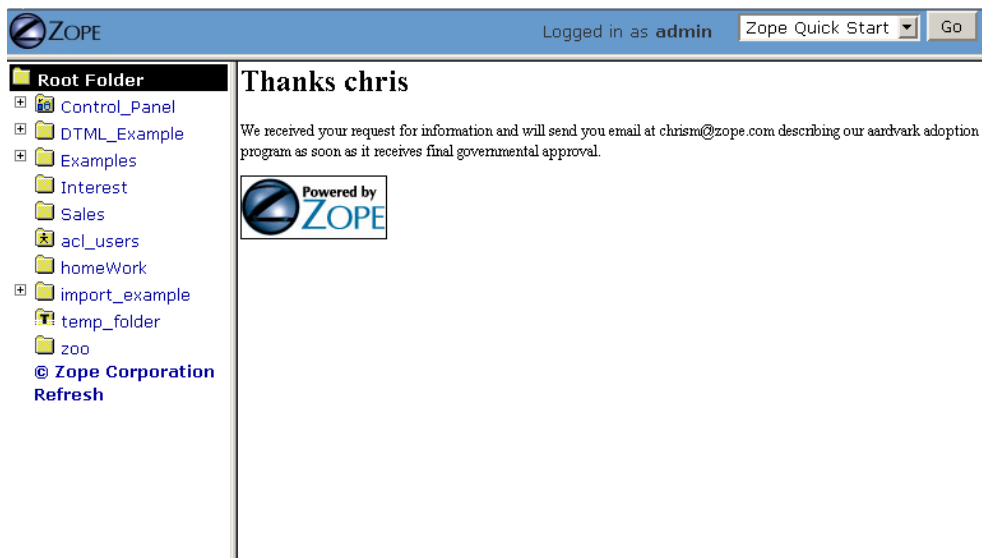


Figure 9-2 Result of submitting the `infoForm` method

The Zope object named `REQUEST` contains information about the current web request. This object is in the DTML name lookup path. The `infoAction` method found the form information from the web request that happened when you clicked the submit button on the rendering of `infoForm`. DTML looks for variables in the current web request, so you can just refer to the form variable names in the target method by name. In our case, we were able to display the values of the form elements `user_name` and `email_addr` in the `infoAction` method just by referring to them by name in their respective `dtml-var` tags. DTML used its lookup rules to search for the variable names. It found the names in the "REQUEST.form" namespace and displayed them. If it had found an object with either name `email_addr` or `user_name` earlier in the lookup (if perhaps there was a Zope object in your acquisition path named `user_name`) it would have found this object first and rendered its results. But, mostly by chance, it didn't, and found the name in `REQUEST` instead.

Let's examine the contents of the Zope `REQUEST` object in order to shed more light on the situation. Create a new DTML Method object named `show_request` in your sandbox folder. Give it the the following body:

```
<dtml-var REQUEST>
```

The `show_request` method will render a human-readable representation of Zope's `REQUEST` object when you click submit on the `infoForm` rendering. Visit the `infoForm` method, and change it to the following:

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

The Zope Book (2.6 Edition)

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

```
<form action="show_request">
name: <input type="text" name="user_name"><br>
email: <input type="text" name="email_addr"><br>
<input type="submit" name="submit" value=" Submit " >
</form>
```

```
<dtml-var standard_html_footer>
```

We changed the form action of the `infoForm` method to `show_request`. Now click the *View* tab of the new `infoForm` method. Fill in some information in the form elements, and click *Submit*. You will see something like the following:

```
form
  submit ' Submit '
  email_addr 'chrism@zope.com'
  user_name 'Chris'

cookies
  tree-s 'eJzTiFZ3hANPW/VYHU0ALlYE1A'

lazy items
  SESSION <bound method SessionDataManager.getSessionData of <SessionDataManager instance at 897d020>

other
  AUTHENTICATION_PATH ''
  user_name 'Chris'
  PUBLISHED <DTMLMethod instance at 8a62670>
  submit ' Submit '
  SERVER_URL 'http://localsaints:8084'
  email_addr 'chrism@zope.com'
  tree-s 'eJzTiFZ3hANPW/VYHU0ALlYE1A'
  URL 'http://localsaints:8084/DTML_Example/show_request'
  AUTHENTICATED_USER admin
  TraversalRequestNameStack []
  URL0 http://localsaints:8084/DTML_Example/show_request
  URL1 http://localsaints:8084/DTML_Example
  URL2 http://localsaints:8084
  BASE0 http://localsaints:8084
  BASE1 http://localsaints:8084
  BASE2 http://localsaints:8084/DTML_Example
  BASE3 http://localsaints:8084/DTML_Example/show_request

environ
  SCRIPT_NAME ''
  HTTP_ACCEPT_ENCODING 'gzip, deflate, compress;q=0.9'
  SERVER_PORT '8084'
  PATH_TRANSLATED '/DTML_Example/show_request'
  HTTP_ACCEPT 'text/xml...'
  GATEWAY_INTERFACE 'CGI/1.1'
  HTTP_COOKIE 'tree-s="eJzTiFZ3hANPW/VYHU0ALlYE1A"'
  HTTP_ACCEPT_LANGUAGE 'en-us, en;q=0.50'
  REMOTE_ADDR '192.168.1.3'
  SERVER_NAME 'saints'
  HTTP_USER_AGENT 'Mozilla/5.0 (Windows; U; Windows NT 5.0; en-US; rv:1.1a+) Gecko/20020629'
  HTTP_ACCEPT_CHARSET 'ISO-8859-1, utf-8;q=0.66, *;q=0.66'
  CONNECTION_TYPE 'keep-alive'
  channel.creation_time 1027876407
  QUERY_STRING 'user_name=Chris&email_addr=chrism%40zope.com&submit=+Submit+'
  SERVER_PROTOCOL 'HTTP/1.1'
  HTTP_KEEP_ALIVE '300'
  HTTP_HOST 'localsaints:8084'
  REQUEST_METHOD 'GET'
  PATH_INFO '/DTML_Example/show_request'
  SERVER_SOFTWARE 'Zope/(unreleased version, python 2.1.3, linux2) ZServer/1.1b1'
  HTTP_REFERER 'http://localsaints:8084/DTML_Example/infoForm'
```

You have instructed the `show_request` method to render the contents of the web request initiated by the `infoForm` method. Note that each section (form, cookies, lazy items, other, and environ) represents a *namespace* inside the REQUEST. DTML searches all of these namespaces for the names you refer to in your `infoForm` form. Note that `email_addr` and `user_name` are in the "form" namespace of the REQUEST. There is lots of information in the rendering of the REQUEST, but for us, this is the most pertinent. For more information on the REQUEST object, visit the Zope Help system, and choose Zope Help -> API Reference -> Request.

Dealing With Errors

Let's perform an experiment. What happens if you try to view the `infoAction` method you created in the last section directly, as opposed to getting to it by submitting the `infoForm` method? Click on the `infoAction` method and then click the `View` tab. You will see results not unlike those in the figure below.

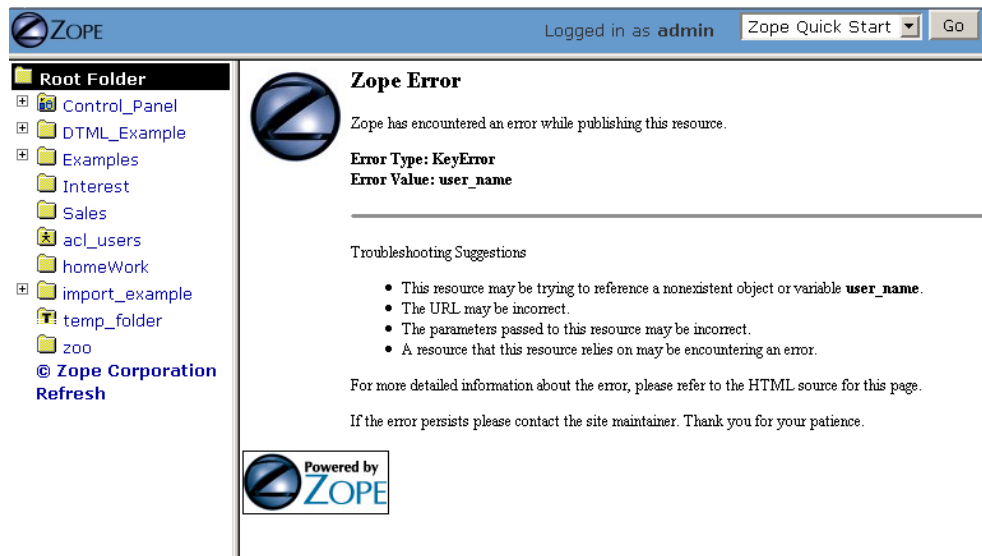


Figure 9-3 DTML error resulting from a failed variable lookup.

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. You can examine the error by visiting the `error_log` object in your root folder. In this case, we know why the error occurred, so visiting the error in the `error_log` isn't really necessary. 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 DTML looks for variables will help you figure out how to fix this kind of problem. In this case, you have viewed a method 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 (its parent folders) 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.

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 the `standard_html_method` object or property is located. Zope will search upwards in the object database 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 new folder in your "sandbox" folder with an id of "Green". Enter the `Green` folder and create a DTML Method with an id of "welcome". Edit the `welcome` DTML Method to have these contents:

```
<dtml-var standard_html_header>
<p>Welcome</p>
<dtml-var standard_html_footer>
```

Now view the `welcome` method. It should look like a simple web page with the word `welcome`, as shown in the figure below.

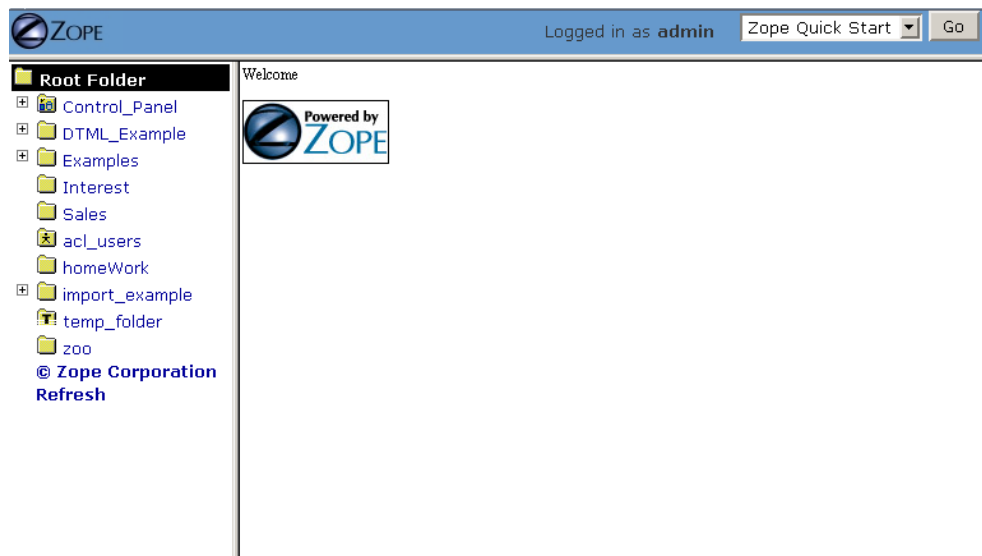


Figure 9-4 Welcome method.

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". Give it the following body:

```
<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. For example, it does not have an ending `</html>` tag. This is just a fragment of HTML that will be used as a header, meant to be included into other pages. This header uses CSS (Cascading Style Sheets) to make some changes to the look and feel of web pages.

Now revisit the `welcome` method and click its *View* tab again. You will see something like the figure below::

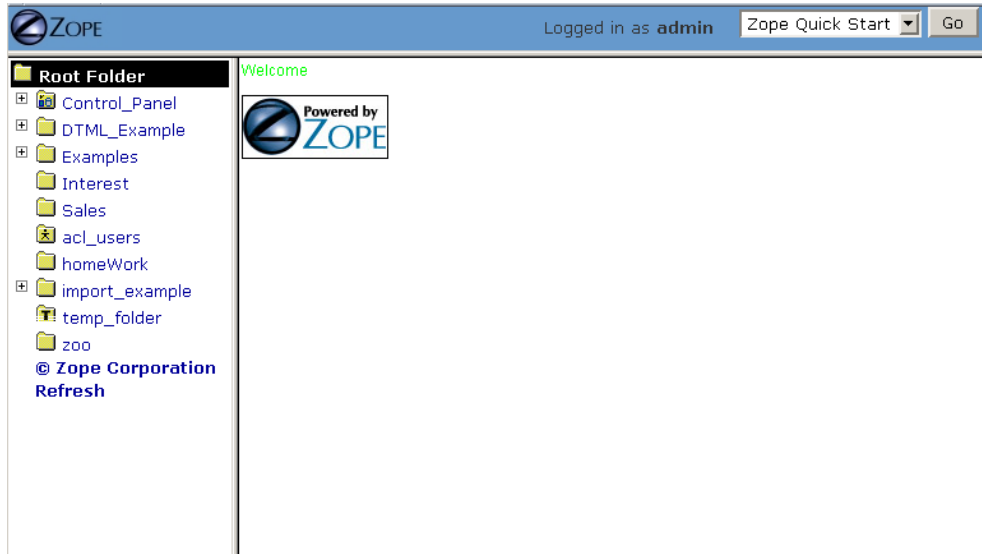


Figure 9-5 Welcome method with custom header.

The rendering 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. You can of course also create a `standard_html_footer`, providing it with local content as well.

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 object which represents the variable and calling it if necessary. We call this basic tag syntax *name* syntax to differentiate it from *expression* syntax.

When you use DTML name syntax, DTML tries to do the right thing to insert the results of the object looked up by the variable name, no matter what that object may be. In general this means that if the variable is another DTML Method or DTML Document, it will be called with appropriate arguments. However, if the variable is *not* another DTML Method or DTML Document, and it requires parameters, you need to explicitly pass the arguments along using an expression.

Expressions used in DTML allow you to be more explicit about how to find and call variables. Expressions are tag attributes that contain small snippets of code in the Python programming language. These are typically referred to as *Python expressions*.

A Python expression is essentially any bit of code that *is not* a Python *statement*. For example, the Python statement `a = 1` assigns "1" to the "a" variable. You cannot use this statement in DTML expressions. Likewise, you cannot use the statement `print "x"` in DTML. It is not an expression. Essentially, an expression must be a combination of values, variables, and Python *operators*. To find out more about Python's expression syntax, see the Python Tutorial at the Python.org web site. For more information specifically about the differences between Python expressions and statements, see the Variables, expressions, and statements chapter of *How To Think Like a Computer Scientist Using Python*.

An expression always results in a return value. For example, the Python expression `"a == 5"` returns the integer 1 if "a" is equal to the integer 5 or the integer 0 if "a" is not equal to the integer 5. The return value of an expression is used by DTML as the *target* of the DTML command.

The primary difference in DTML between using *expressions* as targets and *names* as targets is that DTML does some magic after it locates a *named* targets that it does not do after it finds an expression targets. For example, after finding object with the name `standard_html_header` in the root folder via the name-syntax DTML command `<dtml-var standard_html_header>`, DTML *calls* the `standard_html_header` object, inserting the results into the page. However, when you use an expression-syntax DTML command, like `<dtml-var expr="standard_html_header">`, DTML *will not* call the `standard_html_header` object. Instead it will return a representation of the object as a string. In order to *call* the `standard_html_header` object in an expression-syntax DTML tag, you need to do it explicitly by passing along arguments. When you delve into the realm of DTML expression syntax, DTML "magic" goes away, and you need to become aware of the arguments accepted by the target (if any) and pass them along.

Let's create a Script (Python) object named `getHippo` that *must* be called in DTML with expression syntax, because it takes a non-optional argument that *named* DTML syntax cannot provide.

Create a Script (Python) in your sandbox folder named `getHippo`. Provide it with the following body:

```
## Script (Python) "getHippo"
##bind container=container
##bind context=context
##bind namespace=
##bind script=script
##bind subpath=traverse_subpath
##parameters=trap
##title=
##
return 'The hippo was captured with a %s.' % trap
```

Note that this Script (Python) object takes a single parameter named "trap". It is not an optional parameter, so we need to pass a value in to this script for it to do anything useful.

Now let's make a DTML method to call `getHippo`. Instead of letting DTML find and call `getHippo`, we can use an expression to explicitly pass arguments. Create a DTML method named `showHippo` and give it the following body:

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

Here we've used a Python expression to explicitly call the `getHippo` method with the string argument, `large net`. View the `showHippo` DTML Method. It will return a result not unlike the following:

```
The hippo was captured with a large net.
```

To see why we need to use expression syntax to call this script, let's modify the `showHippo` method to use DTML name syntax:

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

View the method. You will receive an error not unlike the following:

```
Error Type: TypeError
Error Value: getHippo() takes exactly 1 argument (0 given)
```

The `getHippo` method requires that you pass in an argument, `trap`, that cannot be provided using DTML name syntax. Thus, you receive an error when you try to view the `showHippo` method.

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">
```

These two examples if you are to put them in a DTML Method will end up giving you two completely different results. The first example of the DTML `var` tag will automatically *call* the object which is represented by `objectValues`.

In an expression, you have complete control over the variable rendering. In the case of our example, `objectValues` is a method implemented in Python which returns the values of the objects in the current folder. It has no required arguments. So:

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

will call the method. However,

```
<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 Python method which requires no arguments 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.

Common DTML Tags

Below, we discuss the most common DTML tags: the *var* tag, the *if* tag, the *else* tag, the *elif* tag, and the *in* tag, providing examples for the usage of each.

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. Note that as of Zope 2.6, all string values which are retrieved from the REQUEST namespace are HTML-quoted by default. This helps to prevent "cross-site scripting" security holes present in earlier Zope versions, where a user could insert some clever JavaScript into a page in order to possibly make you divulge information to him which could be private. For more information, see the CERT advisory on the topic.

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>
```

Entity-syntax-based DTML tags always "html quote" their renderings. 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" html_quote">">
```

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 within entity-based DTML syntax and many DTML attributes won't work 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 and 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 that has no parameters.:

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

In the chapter entitled Advanced Zope Scripting , we'll look more closely at how to script business logic with Python. 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 chapter entitled Basic Zope Objects . Create a DTML Method with the following body:

```
<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 the figure below.

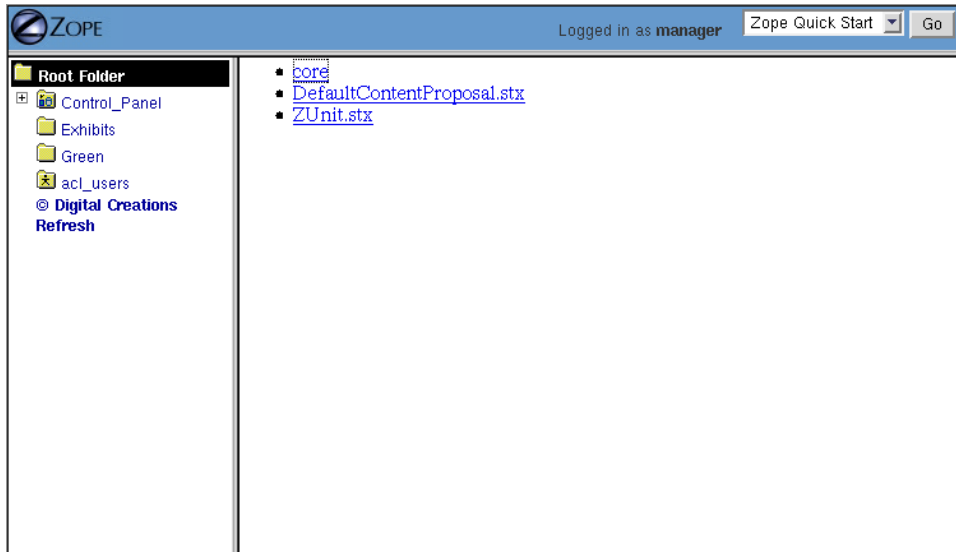


Figure 9-6 Iterating over a list of files.

Let's look at this DTML example step by step. First, the `var` tag is used to insert your common header into the method. 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 the chapter entitled 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.

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 the figure below.

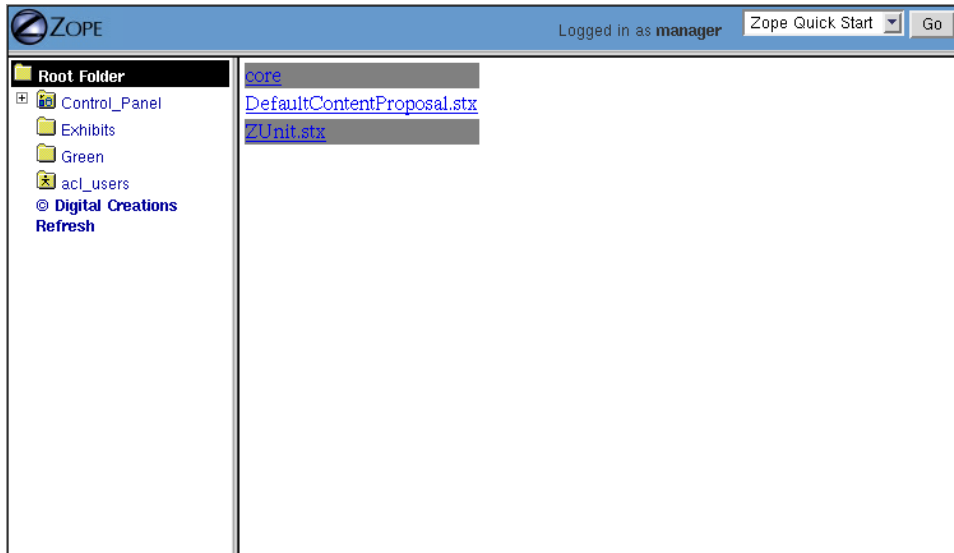


Figure 9-7 File listing with alternating row colors.

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 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 .

Summary

DTML is a powerful tool for creating dynamic content. It allows you to perform fairly complex calculations. In the chapter entitled 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 the chapter entitled Advanced Zope Scripting you'll find out about how to use Python for scripting business logic.

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 features of Page Templates, including how you can use them in your web site to create dynamic web pages easily. In the chapter entitled 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, so you may wonder why we need another template language. First of all, DTML is not aimed at HTML designers. Once an HTML page has been "dynamicized" by inserting DTML into it, the resulting page typically becomes invalid HTML, making it difficult to work with outside Zope. 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. Finally, DTML's namespace model adds too much "magic" to object lookup, without allowing enough control.

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 is not a "dead end". However, it is probable that Page Templates will be used for almost all HTML/XML presentation by Zope Corporation and many members of the Zope community.

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 text contained by 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 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 using a Page Template. 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 edit Page Templates. See the later section in this chapter named "Remote Editing With FTP and WebDAV" 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. Create a Folder to work in named "template_test" in the root of your Zope. Visit this folder and choose "Page Template" from Zope's add list (do *NOT* choose DTML Method or DTML Document, the following examples only work inside a Page Template). 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 a 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. This is because the `tal:replace` statement replaces the entire tag.

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** ." The bold text acts as a placeholder for the dynamic title text. 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 compatibility mode which allows HTML's "loose" markup. In this mode, it's possible to enter "non-well-formed" HTML into a Page Template. However, 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 in order to emit XML. For example, if you wish to emit XHTML, you might put your namespace declarations on the `html` tag:

```
<html xmlns:tal="http://xml.zope.org/namespaces/tal"
      xmlns:metal="http://xml.zope.org/namespaces/metal">
```

For our purposes, we want to emit "loose" HTML, so we leave the *Content-Type* form field as `text/html` and we do not use any XML namespace declarations.

The *Expand macros with editing* control is explained in the chapter entitled 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) specification. For more information on TALES see the Zope Page Templates Reference Appendix .

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 the chapter entitled 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">http://www.example.com</span>.
```

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

```
The URL is http://localhost:8080/template_test/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 `simple_page` template, 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 a `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 you're 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 the chapter entitled *Advanced Page Templates*.

Using the `tal:condition` statement you can check all kinds of conditions. A `tal:condition` statement leaves the tag and its contents in place if its expression has a true value, but removes them 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 in 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 the page so that if the list of objects is empty no table is drawn. Add this to the end of your `simple_page` Page Template:

```
<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>
```

Go and add three Folders named "1", "2", and "3" to the "template_test" folder in which your `simple_page` template lives. Revisit the `simple_page` template and view the rendered output via the *Test* tab. You will see a table that looks much like the below:

Number	Id	Meta-Type	Title
1	simple_page	Page Template	
2	1	Folder	
3	2	Folder	
4	3	Folder	

Note that if the expressions, `container/objectValues` is false (for instance if there are no `objectValues`), 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 table cell in the meta-type column of the above example to look like this:

```
<td>
  <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 "temporary" mock up of a file library page using an HTML "WYSIWYG" ("What You See Is What You Get") editor. Macromedia Dreamweaver, Adobe GoLive, and Netscape Composer are examples of WYSIWYG tools. While you are creating the mockup, just save it to a file on your hard disk.

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>
```

The Zope Book (2.6 Edition)

```
<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's management interface with your browser 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` . For information on creating a new Page Template via an external tool (as opposed to creating one in the ZMI and editing it afterwards with an external tool), see `PUT_factory` in the chapter entitled `Using External Tools` .

Now, with your HTML editor, using FTP, WebDAV (via the DAV "source port"), or HTTP `PUT` , save the above HTML to the URL of the `index_html` Page Template. For example, you may save to the URL `http://localhost:8080/FileLib/index_html` . Different editors have different mechanisms that you can use to do this. See the chapter entitled 'Using External Tools With Zope':`ExternalTools.stx` for more information on using WebDAV, FTP and HTTP `PUT` to communicate with Zope.

****NOTE:** If you're trying to save to Zope via an editor like Netscape Composer or Amaya via HTTP `PUT` (as opposed to FTP or DAV), but you're having problems, try saving the file to `http://localhost:8080/FileLib/index_html/source.html` instead of the URL specified above. Appending `/source.html` to the Zope object name is a "hack" which Page Templates support to get around the fact that HTTP `PUT` attempts to *render* the page before doing the `PUT`, but we actually just want to save the unrendered source. If you're creating an XML file, the "magic" hackaround name is `/source.xml` instead of `/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>
  ...
</head>
<body>
<h1 tal:content="template/title">File Library</h1>
  ...
```

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

Notice the new `content` tag attribute. This attribute says to "replace the *content* of this tag (the text between the `h1` tags) 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/getContentType">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", which is the Python script you created in the current folder (the "container"), which returns a list of Zope objects. The repeat tag causes Zope to create a new table row with columns representing a bit of metadata about each of those objects. 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 data that should go in each cell. During each iteration through the table row loop, the id, the content type, the size, and modification time 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/getContentType`, `item/getSize`, `item/bobobase_modification_time` are all standard API functions that are documented in Zope's online help system as well as in the various appendices to this book.

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 of the default content are very pretty, but the values returned from Zope don't match the format of the dummy content. Instead, they are "raw" numbers. 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>
...
```

Replacing this bit of TAL in your `file_size` template causes Zope to call the 'file_size' script on each object, returning the file's size. When the script runs during the loop, the "context" of the script is "item", which is a File object. This is an example of Zope's *acquisition* in action, as the `file_size` script is actually a *sibling* of the items in the folder, although it can be used as a *method* of the items in the folder. The expression `item/file_size` translates to "find a method of the object `item` named `file_size`. If the object named `file_size` has no "real" method named `file_size`, use acquisition to find a `file_size` method." Zope finds the Script (Python) `file_size` script and use it as a method of the `item` object.

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.month(), date.day())
```

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

```
...
<td tal:content="item/file_date">2001/9/17</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 Macromedia 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. Edit your file with your editor and then save it. When you save it you should use the same URL you used to retrieve it.
3. Optionally reload your page after you edit it, to check for error comments. See the next section for more details on debugging.

You can create new Page Templates without using the Zope Management Interface. See the `PUT_factory` section of the chapter entitled Using External Tools for more information.

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 may have already seen the 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, because they are shown in the "Errors" header of the management interface page when you save the Page Template. 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: PTRuntimeError
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. This information will be available in your *error_log* (in your Zope root folder). 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, but the folder to which you uploaded the template hasn't such a property.

XML Templates

Another example of the flexibility of Page Templates is that they can dynamically render XML as well as HTML. For example, in a chapter within this book entitled *Creating Basic Zope Applications*, you create 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 is 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 in the template text itself. 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. 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 fairly basic and may not provide all of the features or metadata that you need.

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 technologies from and inspired by the CMF.

Creating Basic Zope Applications

XXX - this chapter is not done. I got to just before "Factoring Out Stylesheets" and quit for now. The material prior to that needs to be expanded and cleaned up as well. The examples also need to be converted to page templates.

-chrism

In this chapter you'll learn more about building basic web applications in Zope using Folders, Scripts, and Methods. Another way of terming this is that you'll learn more about creating applications in Zope "instance space".

Building "Instance-Space" Applications

In Zope, there are a few ways to develop a web application. The simplest and fastest way, and the one we've been concentrating on thus far in this book, is to build an application in *instance space*. To understand the term "instance space", we need to once again put on our "object orientation hats".

When you create Zope objects by selecting them from the Zope "Add" list, you are creating *instances* of a *class* defined by someone else (see the Object Orientation chapter if you need to brush up on these terms). For example, when you add a Script (Python) object to your Zope database, you are creating an instance of the Script (Python) class. The Script (Python) class was written by a Zope Corporation engineer. When you select "Script (Python)" from the Add list, and you fill in the form to give an id and title and whatnot, and click the submit button on the form, Zope creates an *instance* of that class in the Folder of your choosing. Instances such as these are inserted into your Zope database and they live there until you delete them.

In the Zope application server, most object instances serve to perform presentation duties, logic duties, or content duties. You can "glue" these instances together to create basic Zope applications. Since these objects are really instances of a class, the term "instance space" is commonly used to describe the Zope root folder and all of its subfolders. "Building an application in instance space" is defined as the act of creating Zope object instances in this space and modifying them to act a certain way when they are executed.

Instance-space applications are typically created from common Zope objects. Script (Python) objects, Folders, DTML Methods, Page Templates, and other Zope services can be glued together to build simple applications.

Instance-Space Applications vs. Products

In contrast to building applications in instance space, you may also build applications in Zope by building them as *Products*. Building an application as a Product differs from creating applications in instance space inasmuch as the act of creating a Product typically allows you to *extend* Zope with new "addable" objects that appear in Zope's "Add" list. Building a Product also typically allows you to more easily distribute an application to other people, and allows you to build objects that may more closely resemble your "problem space". We explore one way to create Products in the chapter entitled Extending Zope. Building a Product is typically more complicated than building an "instance-space" application, so we get started here by describing how to build instance-space applications. When you find that it becomes difficult to maintain, extend, or distribute an instance-space application you've written, it's probably time to reconsider rewriting it as a Product.

Using A Folder as A Container For Your Instance-Space Application

Folders provide containers for your applications. A natural way to build a simple Zope application is to create a Folder in your Zope root folder to hold objects related to the application. For example, you may have an *Invoices* folder to hold an invoice application. You could create "logic" objects inside that folder named *addInvoice* and *editInvoice* to allow you to add and edit the invoices. The actual invoices themselves could be DTML Documents or File objects, which could

also live in the *Invoices* folder. Your *Invoices* folder thus becomes a small application.

URLs are used to work with instance-space Zope applications. As you've seen, you can display a Zope object by visiting its URL in your browser, and in object-orientation terms, when you visit an object in a folder, you are "calling a method in the context of the folder". So for example, the URL `http://localhost:8080/Invoices/addInvoice` calls the `addInvoice` method of the `Invoices` folder. This URL would perhaps take you to a screen that allows you to add an invoice. Likewise, the URL `http://localhost:8080/Invoices/editInvoice?invoice_number=42` might call the `editInvoice` method of the `Invoices` folder, passing it the argument `invoice_number` with a value of 42. The resulting HTML might allow you to edit invoice number 42.

Using Objects as Methods Of Folders Via URLs

The invoices example demonstrates a powerful Zope feature. You can execute a Zope object in the context of a folder by visiting a URL that consists of the folder's URL followed by the id of a Zope object. For example, in the URL `http://localhost:8080/Invoices/addInvoice`, the name `Invoices` refers to a folder. In object-orientation terms, the "final" object in the URL (`addInvoice`) is then used as a "method". The object you call which is used as a method may be a Script (Python) object, a DTML Method, a Page Template, or just about any other kind of Zope object.

This facility is used throughout Zope and is a very general design pattern. In fact you are not restricted to using a folder as the context of a method via a URL. You may call objects as methods in the context of many kinds of Zope objects using the same URL technique.

Using Acquisition In Instance-Space Applications

The Zope facility named Acquisition proves useful when creating instance-space applications. Acquisition allows you to share behavior between different parts of the same application. A folder is said to *acquire* an object by searching for the object in its containers if it cannot find the object by name in itself.

For example, suppose you want to call a method named `viewFolder` on one of your folders. Perhaps you have many different `viewFolder` objects which can be used as methods, each of which represents a particular view of a folder. Zope "figures out" which one you want by first looking in the folder which is named by the "rightmost" portion of the URL. For example, if you invoke the URL `http://localhost:8080/Invoices/July/viewFolder`, and the "Invoices" and "July" objects are folders, the `invoices` object will be searched for a `viewFolder` object first. If Zope can't find the object there it looks for an object named `viewFolder` in the folder's containing folder (`July`). 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.

The Special Folder Object `index_html`

If there is an object in a Zope folder named `index_html`, the return value of this object will be used as the default view of the folder when the folder's URL is called. This is analogous to how an `index.html` file provides a default view for a directory in Apache and other web servers. Instead of explicitly including the name `index_html` in your URL to show default content for a folder, you can omit it. 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 in the *Invoices* folder and display its results. You can also use the more explicit URL `http://localhost:8080/Invoices/index_html`, and it will display the same content.

A folder can also *acquire* an `index_html` object from its parent folders. You can use this behavior to create a default view for a set of folders. To do so, create an `index_html` object in a folder which contains another set of folders. This default view will be used for all the folders in the set. This behavior is already evident in Zope. If you create a set of empty Folders in the Zope root folder, you will notice that when you view any of the Folders via a URL, the content of

the "root" folder's *index_html* method is displayed. The *index_html* in the root folder is acquired. Furthermore, if you create more empty folders inside the folders you've just created in the root folder, a visit to these folders' URLs will also show the root folder's *index_html*. This is acquisition at work. NOTE: We are using the *index_html* method as an example here, but this will work with any Zope object which acts as a method, it needs not be named "index_html".

If you want a different default view of a given folder, just create a custom *index_html* object in that particular folder. This allows you to override the default view of a particular folder on a case-by-case basis, while allowing other folders defined at the same level to acquire a common default view.

The *index_html* object may be a DTML Method, a Page Template, a Script (Python) object, or any other Zope object that is URL-accessible and which returns browser-renderable content. The content is typically HTML, but Zope doesn't care. You can spit out XML or text or whatever you like.

Building the Zope Zoo Website

In this section, we'll create a simple web site in instance space 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, you will need to create some basic site structure. We need to create some folders in your Zope system that represent the structure of your site. Let's use a zoo structure made out of Folders, as shown in the figure below.

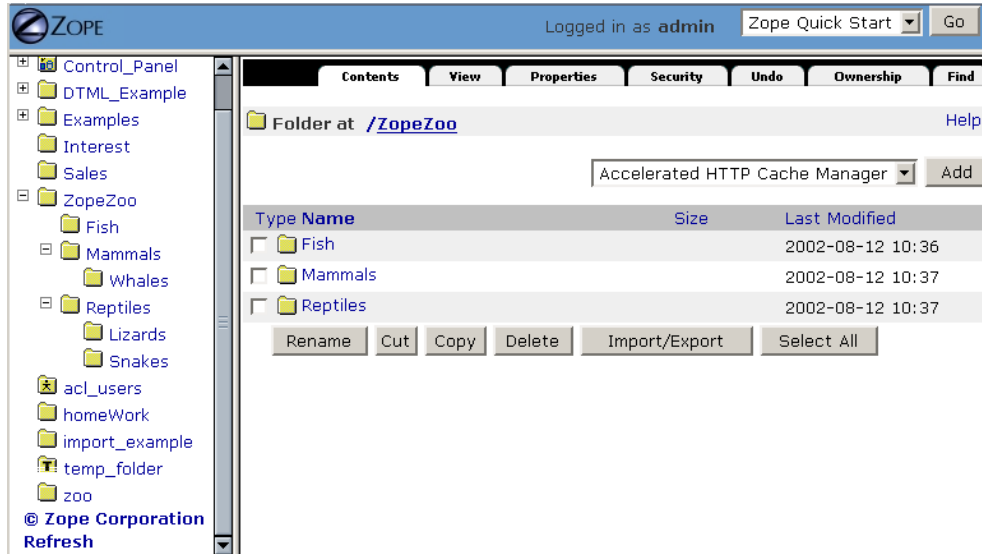


Figure 5-1 Zoo folder structure.

You should create a top-level folder named *ZopeZoo* . Within the *ZopeZoo* folder, you should create three subfolders, *Reptiles* , *Mammals* and *Fish* . Within the *Mammals* folder, you should create a folder named *Whales* . Within the *Reptiles* folder, you should create two folders, *Lizards* and *Snakes* .

To navigate your site, users will visit the default view of the *ZopeZoo* folder (the "front 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. For instance, if the user wishes to visit the "Mammals" section, the view of the Mammals section should have a similar interface to that of the Zoo itself. Also, the user should be able to back out of a section and go up to the parent section.

To provide navigation facilities, in the *ZopeZoo* folder, create a DTML Method named *navigation* :

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

When the method you just created is executed, it displays a list of links. Each link targets the default view of a subfolder. The list of subfolders displayed depends on the context in which the method is executed. For example, if the method is executed in the context of the "Mammals" folder, it will display a link to the default view of the Whales folder. If the method is executed in the context of the "ZopeZoo" folder, it will display links to the default views of the "Mammals", "Fish", and "Reptiles" folders. It's important to notice that this method can be used to display the contents of *any* folder, so we can use it for most of our "default" folder views. Furthermore, since we've placed this method in the *ZopeZoo* folder, each of the zoo subfolders will acquire and use it.

Now, you need to incorporate the navigation method into the site. Let's create two DTML methods. One will be used as a standard "header" for all pages within the site, the other a standard "footer". Do this by first creating a DTML Method named *standard_html_header* in the *ZopeZoo* folder. We will include the navigation links in the display of this method by referencing the *navigation* method via 'dtml-var':

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

Now create a DTML Method named *standard_html_footer* in your *ZopeZoo* folder and provide it with this content:

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

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

In order to display our navigation and standard header and footer, we need a front page that serves as the welcome screen for Zoo visitors. In order to do so, create a DTML Method in the *ZopeZoo* folder named *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 of the *ZopeZoo* folder. The results of doing so are shown in the figure below.



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 included 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 instead of from the `ZopeZoo` 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 `ZopeZoo` 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 view the `ZopeZoo` folder to see how this new link works, as shown in the figure below.



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="id != 'ZopeZoo'">
  <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 context object is named `ZopeZoo` and declines to display the "Return to parent" link if so. View the *ZopeZoo* folder to see the result.

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 in the Zoo:

```
<dtml-if expr="id != 'ZopeZoo'">
  <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 *ZopeZoo* folder again and navigate into the *Reptiles* section. Notice that within the *Reptiles* section, you see a header which says "Reptiles Section", as shown in the figure below.



Figure 5-4 Zoo page with section information.

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>
```

Anonymous User - June 16, 2002 6:47 pm:

The title is missing from the `standard_html_header` for no obvious reason.

It was there in the initial code for `standard_html_header`, so it should be present here as well, IMHO.

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 .



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.

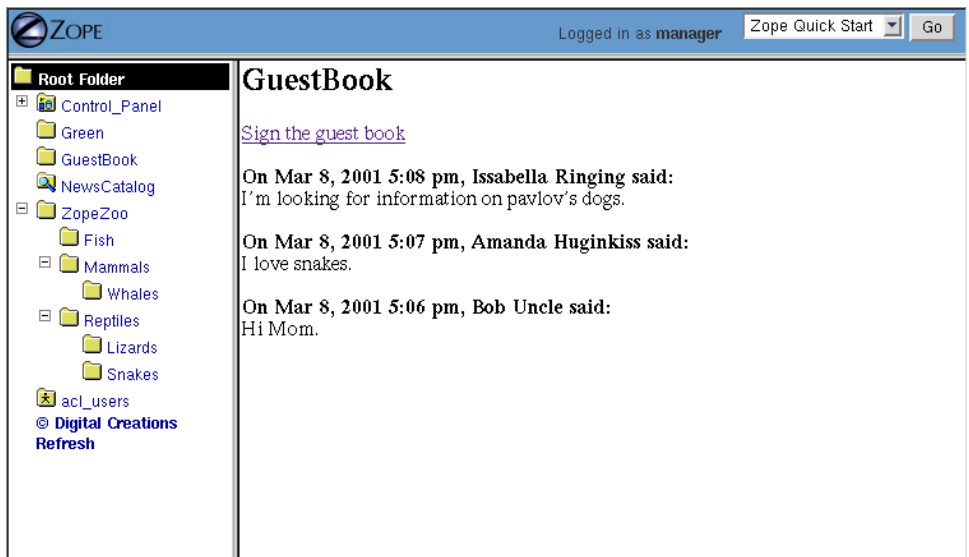


Figure 5-6 Zoo guest book.

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 Files, one file for each guest book entry. When a new entry is added to the guest book, a new file is created in the *GuestBook* folder. To delete an unwanted entry, just go into the *GuestBook* folder and delete the unwanted file 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 File in the folder starting with
the newest documents first. -->

<dtml-in expr="objectValues('File')"
          sort="bobobase_modification_time" reverse>

<!-- Display the date, author and contents of each file -->

  <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 files in the folder and displays each one. Notice that this method assumes that each file 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 file 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>
```

The Zope Book (2.6 Edition)

```
<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>
```

```
Anonymous User - May 9, 2002 4:48 pm:
URL1? Not working for me!?
```

```
Anonymous User - June 3, 2002 2:24 am:
```

```
I've tried to send some comments to the guestbook, files of comments have been created but those comments didn't list out like the above pic.
```

```
Anonymous User - June 13, 2002 5:01 am:
```

```
is it possible to send the form data straight to the form and then return to the guestbook page with the validated entry?
```

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 file 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 file id
id='entry_%d' % len(context.objectIds())

# create the file
context.manage_addProduct['OFSP'].manage_addFile(id,
                                                    title="", file=comments)

# add a guest_name string property
doc=getattr(context, id)
doc.manage_addProperty('guest_name', guest_name, 'string')
```

```
Anonymous User - May 22, 2002 11:06 am:
```

```
Where does the ['OFSP'] come from?
```

```
After digging around and doing a few searches, I found it's a core part of Zope, but no explanations.
```

```
Anonymous User - June 12, 2002 12:38 pm:
```

```
I get an error that states:
```

```
Error Type: TypeError
```

```
Error Value: addEntry() takes no arguments (2 given)
```

```
What did I do wrong?
```

```
Anonymous User - June 15, 2002 9:54 am:
```


You forgot to include the parameters for the Python script in the Parameters field when pasting the script into a new Python Script object. You need to explicitly state what parameter your python 'function' takes when creating the script.

This script uses Zope API calls to create a File and to create a property on it. This script performs the same sort of actions in a script that you could do manually; it creates a file, 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 files. 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 file. You can do this by setting the *Proxy role* of the script to *Manager*. This means that when the script 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* script, as shown in Figure 5-7.

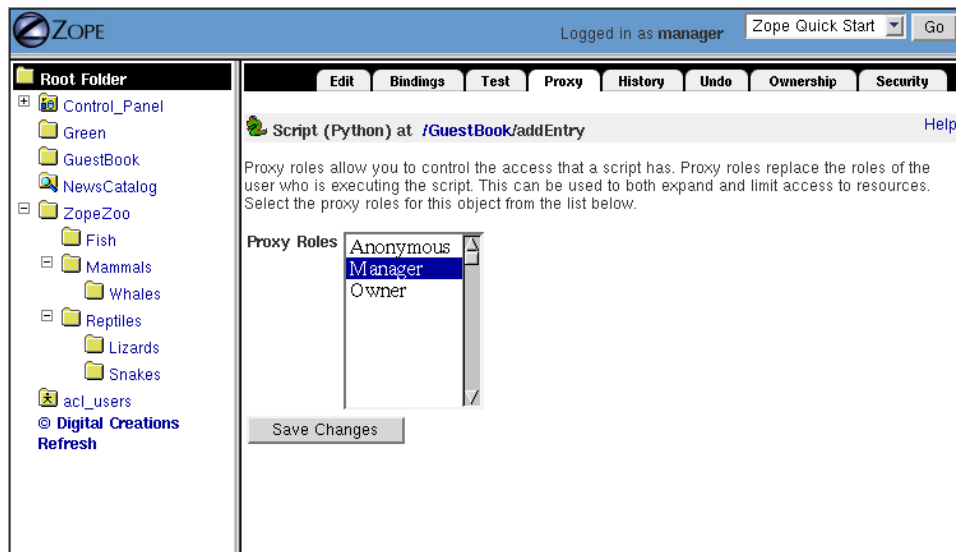


Figure 5-7 Setting proxy roles for the addEntry script.

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>
```

The Zope Book (2.6 Edition)

```
<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>
```

Anonymous User - May 4, 2002 1:12 am:

The guestbook entries are not 'DTML Document' type objects, but 'File' type objects in the above examples. Thus, I think the following codes are more suitable.

```
<?xml version="1.0" encoding="EUC-KR" ?>
```

```
<guestbook>
  <dtml-in expr="objectValues('File')">
    <entry>
      <author><dtml-var guest_name null="Anonymous"></author>
      <comments><dtml-var sequence-item 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.

Users and Security

Introduction to Zope Security

Zope is a multi-user system. However, instead of relying upon the user accounts provided by the operating system under which it runs, Zope maintains one or more of its own user databases. It is not necessary to create a user account on the operating system under which Zope runs in order to grant someone a user account which they may use to access your Zope application or manage Zope via its management interface.

It is important to note that Zope users do not have any of the privileges of a "normal" user on your computer's operating system. For instance, they do not possess the privilege to change arbitrary files on your computer's filesystem. Typically, a Zope user may influence the content of databases that are connected to Zope may execute scripts (or other "logic" objects) based on Zope's security-restricted execution environment. It is also possible to allow users to create their own scripts and content "through the web" by giving them access to the Zope Management Interface. However, you can restrict the capability of a user or a class of users to whatever suits your goals. The important concept to absorb is that Zope's security is entirely divorced from the operating system upon which it runs.

In Zope, users have only the capabilities granted to them by a Zope *security policy*. As the administrator of a Zope system, you have the power to change your Zope system's security policies to whatever suits your business requirements.

Furthermore, using security policies you can provide the capability to "safely" *delegate* capabilities to users defined within different parts of a Zope site. "Safe delegation" is one of the important and differentiating features of Zope. It is possible to grant users the capability in a Zope site to administer users and create scripts and content via the Zope Management Interface. This is called "safe" delegation because it is relatively "safe" to grant users these kinds of capabilities within a particular portion of a Zope site, as it does not compromise operating system security nor Zope security in other portions of the site. Caveats to safe delegation pertain to denial of service and resource exhaustion (it is not possible to control a user's resource consumption with any true measure of success within Zope), but it is possible to delegate these capabilities to "semi-trusted" users in order to decentralize control of a web site, allowing it to grow faster and require less oversight from a central source.

In this chapter we will look more closely at administering users, building roles, mapping roles to permissions, and creating a security policy for your Zope site.

Review: Logging In and Logging Out of the Zope Management Interface

As we first saw in the chapter entitled *Installing Zope*, you may log into the Zope Management Interface by visiting a "management" URL in your web browser, entering a username and password when prompted. We also pointed out in *Using the Zope Management Interface* that due to the way many web browsers work, you often must perform an extra step when an authentication dialog is raised or you must quit your browser to log out of Zope. Review these chapters for more information about the basics of logging in and out of the Zope Management Interface.

Zope's "Stock" Security Setup

"Out of the box", a vanilla Zope site has two different classes of users: *Managers* and *Anonymous* users. You have already seen via the *Installing Zope* chapter how you can log into the Zope management interface with the "initial" user called "admin". The initial "admin" user is a user with the *Manager* role, which allows him to perform almost any duty that can be performed within a Zope instance.

By default, in the "stock" Zope setup, Managers have the rights to alter Zope content and logic objects and view the management interface, while the Anonymous users are only permitted to view rendered content. This may be sufficient for many simple websites and applications, especially "public-facing" sites which have no requirement for users to "log in" or compose their own content.

Identification and Authentication

When a user accesses a protected resource (for example, by attempting to view a "protected" DTML Method) Zope will ask the user to log in by presenting some sort of authentication dialog. Once the dialog has been "filled out" and submitted, Zope will look for the user account represented by this set of credentials.

Zope *identifies* a user by examining the username and password provided during the entry into the authentication dialog. If Zope finds a user within one of its user databases with the username provided, the user is identified.

Once a user has been identified, *authentication* may or may not happen. Authentication succeeds if the password provided by the user in the dialog matches the password registered for that user in the database.

Zope will only attempt to identify and authenticate a user if he attempts to perform an action against Zope which an anonymous user has not been permitted the capability to perform; if a user never attempts to access a protected resource, Zope will continue to treat the user as an anonymous user.

Zope prompts a user for authentication if the user attempts to access a "protected" resource without an adequate set of credentials, as determined by the resource's security policy. For example, if a user attempts to access a method of an object which has a restrictive security policy (like all of Zope's management interface methods) the user will be prompted for authentication if he is not logged in. You've seen this behavior already if you've ever attempted to log in to Zope and have been asked for a username and password to access the ZMI. The ZMI is an example of a Zope application. Zope's security machinery performs security checks on behalf of the ZMI; it "pops up" an authentication dialog requesting that the user enter a username and password.

Different things can happen with respect to being prompted for authentication credentials in response to a request for a protected resource depending on the current state of a login session. If the user has not yet logged in, Zope will prompt the user for a username and password. If the user is logged in but the account under which he is logged in does not have sufficient privilege to perform the action he has requested, Zope will prompt him for a *different* username and password. If he is logged in and the account under which he has logged in *does* have sufficient privileges to perform the requested action, the action will be performed. If a user cannot be authenticated because he provides a nonexistent username or an incorrect password to an existing authentication dialog, Zope re-prompts the user for authentication information as necessary until the user either "gets it right" or gives up.

In general, there is no need for a user to log in to Zope if he only wishes to use public resources. For example, to view the parts of your Zope website that are publically available, a user should not need to log in.

Authorization, Roles, and Permissions

Once a user has been authenticated, Zope determines whether or not he has access to the resource which is being protected. This process is called *authorization*. Remember that the only reason that Zope asked for credentials is because the user was attempting to view a resource which was not viewable by an anonymous user. The "resource which is being protected" referred to above is the object which the user requested to perform an action against, which caused the authentication process to begin.

The process of authorization involves two intermediary layers between the user and the protected resource: *roles* and *permissions*.

Users have *roles* which describe "what they can do" such as "Author", "Manager", and "Editor". These roles are controlled by the Zope system administrator. Users may have more than one role, and may have a different set of roles in different contexts. Zope objects have permissions which describe "what can be done with them" such as "View", "Delete objects", and "Manage properties". These permissions are defined either within Zope itself or by Zope *Products*, each of which may define its own set of permissions.

A *context* in Zope is a "place" within the Zope object hierarchy. In relation to security, a context is an object that has a location within the Zope Object Database. For example, a description of a context could be expressed as "the `Examples` Folder object within the Zope root object". Another example of a context might be "a DTML Method object named `show_css` within the Zope root folder". In essence, a context can be thought of as an object's "location" within the Zope Object Database, described by its "path". Each object that exists in the Zope Object Database which has a web-manageable interface can be associated with its own security policy. Objects can also "acquire" security policies from containing objects in order to ease the burden of creating a security policy. In fact, most Zope objects acquire their security policies from their containers because it makes a given security policy easier to maintain. Only when there are exceptions to the "master" security policy in a context are individual objects associated with a differing policy.

In essence, *security policies map roles to permissions in a context*; in other words they say "who" can do "what", and "where". For example, the security policy for a Folder (the context) may associate the "Manager" role (the roles) with the "Delete objects" permission (the permissions). Thus, this security policy allows managers to delete objects in this folder. If objects created within this folder do not override their parents' security policy, they acquire this policy. So, for example, if a DTML Method is created within this folder, it may also be deleted by users with the Manager role. Subobjects within subfolders of the original folder have the same policy unless they override it themselves, ad infinitum.

Managing Users

In the chapter entitled *Installing Zope*, you were provided with an "initial" account named `admin`, which possesses the `Manager` role, allowing you to manage the objects in your Zope instance. To allow other people to log into Zope, and to further understand Zope security, you should create user accounts under which different users may authenticate.

Creating Users in User Folders

A Zope *User* object defines a user account. A Zope *User* has a name, a password, one or more *roles*, and various other properties. Roles are granted to a user in order to make it easier to control the scope of what he or she may do within a Zope site.

To create user accounts in Zope, you create users within *User Folders*. A user folder contains user objects that define Zope user accounts. User Folder objects always have a Zope "id" of `acl_users`. More than one user folder can exist within a Zope instance, but more than one user folder may not exist within the *same* Zope Folder.

To create a new account, visit the root Zope folder. Click on the object named `acl_users`. Click the *Add* button to create a new user.

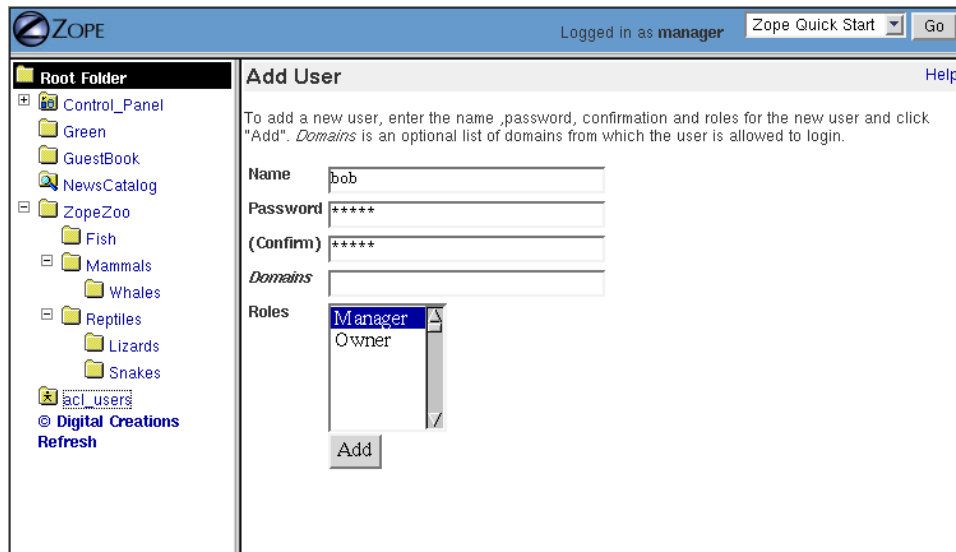


Figure 11-1 Adding a user to a user folder.

The form shown above lets you define the user. Type a username in the *Name* field (for example, "bob"). The username can contain letters, spaces, and numbers. The username is case sensitive. Choose a password for your new user and enter it in the *Password* and *(Confirm)* fields. In the next section, we will provide information about allowing a user to change his or her own 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 a user to log in from work you could enter your work's Internet domain name, 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 wildcard names 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* multiple select list indicates which roles the user should have. The Zope default roles include *Manager* and *Owner* . In general users who need to perform management tasks using the Zope Management Interface should be given the *Manager* role. The *Owner* role is not appropriate to grant in most cases because a user normally only has the *Owner* role in the context of a specific object. Granting the *Owner* role to a user in the User Folder management interface grants that user ownership of all objects within the folder in which the user folder is placed as well as all subfolders and subobjects of that folder. It is unfortunate that the *Owner* role is present in the list of roles to choose from in the User Folder management interface, as it is confusing, little-used, and only now exists to service backwards compatibility. In most cases it can be ignored completely.

You may define your own roles such as *Editor* and *Reviewer* . In the section later in this chapter named "Defining Roles", we will create a new set of roles. For now, we will work with the "stock" Zope roles.

To create the new user click the *Add* button. You should see a new user object in the user folder.

Zope User accounts defined in the "stock" user folder implementation do not support additional properties like email addresses and phone numbers. For support of properties like these, you will have to use external User products like the CMF Membership Component (in the CMF) or exUserFolder .

Users can not be copied and pasted between User Folders. The facility does not exist to perform this.

Editing Users

You can edit existing users by clicking on their name within the User Folder management interface screen. Performing this action causes a form to be displayed which is very similar to the form you used to create a user. In fact, you may control most of the same settings that we detailed in the "Adding Users" section from within this form. It is possible to visit this management screen and change a user's password, his roles, and his domain settings. In the "stock" user folder implementation, you cannot change a user's name, however, so you will need to delete and recreate a user if you need to change his name.

It is not possible for someone to find out a user's password by using the management interface. Another manager may have access to *change* another user's password, but he may not find out what the current password is from within the management interface. If a user's password is lost, it is lost forever.

Like all Zope management functions, editing users is protected by the security policy. Users can only change their password if they have the *Manage Users* permission in the context of their own user folder, which managers have by default. It is often desirable to allow users to change their own passwords. One problem is that by giving a user the *Manage Users* permission, they are also able to edit other user accounts and add/delete users. This may or may not be what you want.

To grant the capability for users to change their own passwords without being able to influence other users' information, set up a script with *Proxy Roles* to do the work for you. See msx's mini-how-to for more information, or create a script to do so after reading the section within this chapter entitled "Proxy Roles".

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.

Defining a User's Location

Zope can contain multiple user folders at different locations in the object database hierarchy. A Zope user cannot access protected resources above the user folder in which their account is defined. The location of a user's account information determines the scope of the user's access.

If an account is defined in a user folder within the root folder, the user may access protected objects defined within the root folder. This is probably where the account you are using right now is defined. You can however, create user folders within any Zope folder. If a user folder is defined in a subfolder, the user may only access protected resources within that subfolder and within subfolders of that subfolder, and so on.

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 access protected Zope resources above the folder at */BeautySchool/Hair* . Effectively Ralph's view of protected resources in 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. If Ralph was defined as having the *Manager* role, he would be able to go directly to */BeautySchool/Hair/manage* to manage his resources, but could not access */BeautySchool/manage* at all.

To access the Zope Management Interface as Manager user who is *not* defined in the "root" user folder, use the URL to the folder which contains his user folder plus *manage* . For example, if Ralph Scissorhands above has the Manager role as defined within a user folder in the *BeautySchool/Hair* folder, he would be able to access the Zope Management Interface by visiting <http://zopeserver/BeautySchool/Hair/manage> .

Of course, any user may access any resource which is *not* protected, so a user's creation location is not at all relevant with respect to unprotected resources. The user's location only matters when he attempts to use objects in a way that requires authentication and authorization, such as the objects which compose the Zope Management Interface.

It is straightforward to delegate responsibilities to site managers using this technique. 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. By doing so, you "safely" *delegate* the responsibility for these objects to these users.

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.

Delegation is a very common pattern 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.

Working with Alternative User Folders

It may be that you don't want to manage your user account through the web using Zope's "stock" user folder implementation. Perhaps you already have a user database, or perhaps you want to use other tools to maintain your account information. Zope allows you to use alternate sources of data as user information repositories. You can find an ever-growing list of alternate user folders at the Zope web site Products area . Here is a sampling of some of the more popular alternative user folders available.

Extensible User Folder — `exUserFolder` allows for authentication from a choice of sources and separate storage of user properties. It has been designed to be usable out of the box, and requires very little work to set up. There are authentication sources for PostgreSQL, RADIUS and SMB and others as well as normal ZODB storage.

etcUserFolder — This user folder authenticates using standard Unix `/etc/passwd` style files.

LDAP User Folder — 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.

MySQLUserFolder — This user folder authenticates from data within a MySQL database.

Some user folders provide alternate login and logout controls in the form of web pages, rather than relying on Basic HTTP Authentication 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" who possess no credentials. This is 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` additionally possesses the `Anonymous role`. The "stock" Zope security policy restricts users which possess the `Anonymous` role from accessing nonpublic resources. 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 attempt authentication. Even if you've got a user account on the system, Zope will consider you the `Anonymous` user until you been prompted for login and you've successfully 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 typically only useful for two things: fixing broken permissions, and creating and changing user accounts.

You may use the emergency user account to create or change other user accounts. Typically, you use the emergency user account to define accounts with the `Manager` role or change the password of an existing account which already possesses the `Manager` role. This is useful in case you lose your management user password or username. Typically, after you create or change an existing a manager account you will log out as the emergency user and log back in as the manager.

Another reason to use the emergency user account is to "fix" broken permissions. If you lock yourself out of Zope by removing permissions you need to manage Zope, you can use the emergency user account to repair the permissions. 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 with respect to the object you're attempting to view. Then log out and log back with your manager account and you should have enough access to fix anything else that is broken.

The emergency user cannot create new "content", "logic" or "presentation" objects. A common error message seen by users attempting to use the emergency user account in trying to create a new object is shown below.

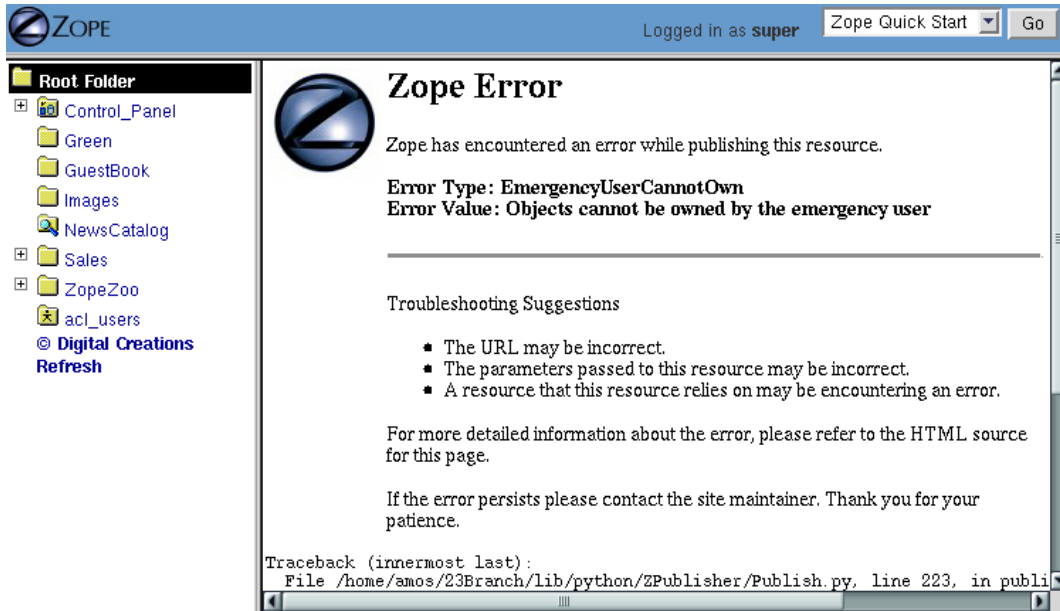


Figure 11-2 Error caused by trying to create a new object when logged in as the emergency user.

The error above lets you know that the emergency user cannot create new objects. This is "by design", and the reasoning behind this policy may become clearer later in the chapter when we cover ownership.

Creating an Emergency User

Unlike normal user accounts that are defined through the Zope Management Interface, the emergency user account is defined through a file in the filesystem. You can change the emergency user account by editing or generating the file named `access` in the Zope home directory (the main Zope directory). Zope comes with a command line utility in the Zope home directory named `zpasswd.py` to manage the emergency user account. On UNIX, run `zpasswd.py` by passing it the `access` file path as its only argument:

```
$ cd (... where your ZOPE_HOME is... )
$ 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:
```

Due to pathing differences, Windows users usually need to enter this into a command prompt to invoke `zpasswd`:

```
> cd (... where your ZOPE_HOME is ...)
> cd bin
> python ..\zpasswd.py ..\access
```

The `zpasswd.py` script steps you through the process of creating an 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. When setting up or changing the emergency user's details, you need to restart the Zope

process for your changes to come into effect.

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.

Initial users are defined in a similar way to the emergency user; they are defined in a file on the filesystem named `inituser`. On UNIX, the `zpasswd.py` program can be used to edit or generate this file the same way it is used to edit or generate the emergency user `access` file:

```
$ cd ( ... were your ZOPE_HOME is ... )
$ 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 an `inituser` file which contains a user named "bob" and will set its password. The password is not echoed back to you when you type it in. The effect of creating an `inituser` file depends on the state of the existing Zope database.

When Zope starts up, if there are *no* users in the root user folder (such as when you start Zope with a "fresh" ZODB database), and an `inituser` file exists, the user defined within `inituser` will be created within the root user folder. If any users already exist within the root user folder, the existence of the `inituser` file has no effect. Normally, initial users are created by the Zope installer for you, and you shouldn't have to worry about changing them. Only in cases where you start a new Zope database (for example, if you delete the `var/Data.fs` file) should you need to worry about creating an `inituser` file. Note that if Zope is being used in an `INSTANCE_HOME` setup, the created "inituser" file must be copied to the `INSTANCE_HOME` directory. Most Zope setups are not `INSTANCE_HOME` setups (unless you've explicitly made it so), so you typically don't need to worry about this. The `inituser` feature is a convenience and is rarely used in practice except by the installer.

Protecting Against Password Snooping

The HTTP Basic Authentication protocol that Zope uses as part of its "stock" user folder implementation passes login information "over the wire" in an easily decryptable way. It is employed, however, because it has the widest browser support of any available authentication mechanism.

If you're worried about someone "snooping" your username/password combinations, or you wish to manage your Zope site ultra-securely, you should manage your Zope site via an SSL (Secured Sockets Layer) connection. The easiest way to do this is to use Apache or another webserver which comes with SSL support and put it "in front" of Zope. Some (minimalistic) information about setting up Zope behind an SSL server is available at Unfo's member page on Zope.org , on Zopelabs.com . The chapter of this book entitled Virtual Hosting also provides some background that may be helpful to set up an SSL server in front of Zope.

Managing Custom Security Policies

Zope security policies control authorization; they define *who* can do *what* and *where* they can do it. Security policies describe how roles are associated with permissions in the context of a particular object. 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 generalization makes your security policies simple and more powerful. Of course, you can make exceptions to your policy for specific users, actions, and objects.

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, each Zope user can have one or more roles.

Roles make it easier for administrators to manage security. Instead of forcing an administrator to specifically define the actions allowed by each user in a context, the administrator can define different security policies for different user roles in a context. Since roles are classes of users, he needn't associate the policy directly with a user. Instead, he may associate the policy with one of the user's 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 whom have provided valid authentication credentials. This role means that Zope "knows" who a particular user is. When Users are logged in they are considered to also have the Authenticated role, regardless of other roles.

For basic Zope sites you can typically "get by" with only having `Manager` and `Anonymous` roles. For more complex sites you may want to create your own roles to classify your users into different categories.

Defining Global Roles

A "global" role is one that shows up in the "roles" column of the `Security` tab of your Zope objects. To create a new "global" role go to the `Security` tab of your root Zope object (or any other `folderish` Zope object) 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 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 at which they are defined and "below" in the object hierarchy. For example, if you create a role in the `Examples` folder that exists in the Zope root folder, that role cannot be used outside of the `Examples` folder and any of its subfolders and subobjects. 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.

Understanding Local Roles

Local roles are an advanced feature of Zope security. Specific *users* can be granted extra roles when working within the context of a certain object by using a local role. If an object has local roles associated with a user then that user gets those additional roles while working with that object, without needing to reauthenticate.

For example, if a user creates an object using the Zope Management Interface, they are always given the additional local role of *Owner* in the context of that object. A user might not have the ability to edit DTML Methods in general if he does not possess a set of global roles which allow him to do so, but for DTML Methods he owns, the user may edit the DTML Method by virtue of possessing the *Owner* local role.

Local roles are a fairly advanced security control. Zope's automatic control of the *Owner* local role is likely the only place you'll encounter local roles unless you create an application which makes use of them. 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

A permissions defines a single action which can be taken upon a Zope object. 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. Permissions are defined by Zope *Product* developers and the Zope "core" itself. *Products* are responsible for creating a set of permissions which are relevant to the types of objects they expose.

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.

The default Zope permissions are described in appendix A of the Zope Developer's Guide

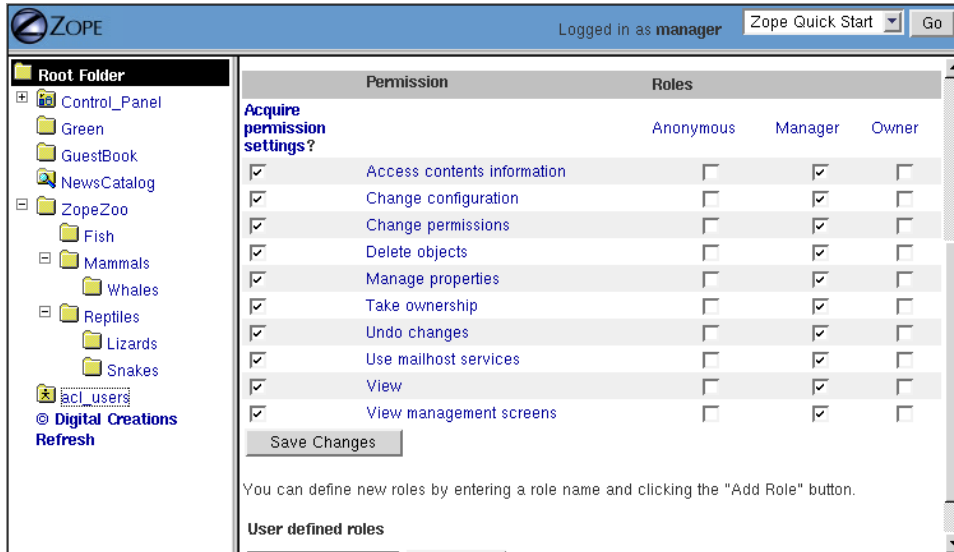


Figure 11-3 Security settings for a mail host object.

As you can see in the figure above, a mail host has a limited palette 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 a security policy on almost any Zope object. To set a security policy on an object, go the object's *Security* tab. For example, click on the security tab of the root folder.

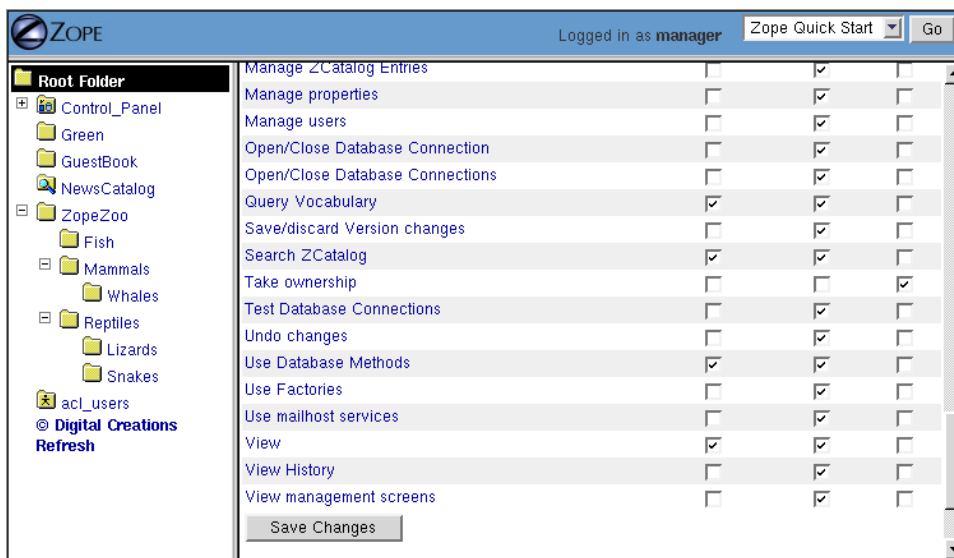


Figure 11-4 Security policy for the root folder.

In the figure above, the center of the screen displays 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 in the context of the object being managed. In this case, the context is the root folder.

Many Zope Products add custom security permissions to your site when you install them. This can make the permissions list grow quite large, and unwieldy. Product authors should take care to re-use suitable existing permissions if possible, but many times it's not possible, so the permission list grows with each new Product that is installed.

You'll notice by virtue of visiting the Security tab of the root folder that Zope comes with a default security policy that allows users which possess the `Manager` role to perform most tasks, and that allows anonymous users to perform only a few restricted tasks. The simplest (and most effective) way to tailor this policy to suit your needs is to change the security settings in the root folder.

For example, you can make your site almost completely "private" by disallowing anonymous users the ability to view objects. 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*. We explored acquisition in the Acquisition chapter. Zope security makes extensive use of 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.

Suppose you want to make a folder private. As we saw before this merely requires denying the *Anonymous* role the *View* permission in the context of this object. But even though the "View" permission's box may be unchecked the folder might not be 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.

Each checked checkbox gives a role permission to do an action or a set of actions. With *Acquire permission settings* checked, these permissions are *added* to the actions allowed in the parent folder. If *Acquire permission settings* is unchecked on the other hand, checkboxes must be explicitly set, and the security setting of the parent folder will have no influence.

In general, you should always acquire security settings unless you have a specific reason to not do so. This will make managing your security settings much easier as much of the work can be done from the root folder.

Security Usage Patterns

The basic concepts of Zope security are simple: roles and permissions are mapped to one another to create security policies. Users are granted roles (either global roles or local roles). User actions are restricted by the roles they possess in the context of an object. 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 almost 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

The pattern of *delegation* is very central to Zope. Zope encourages you to collect like resources in folders together and then to create user accounts in these folders to manage 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 changing any objects which live outside the *Sales* folder, so you don't need to add him to the *acl_users* folder in the root folder. Instead, you would 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.zopezo.org/Sales/manage> .

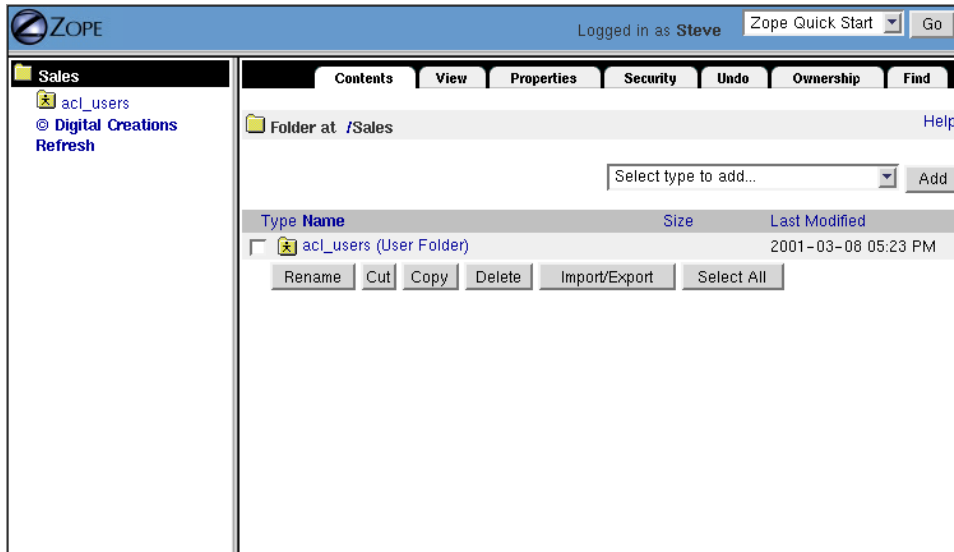


Figure 11-5 Managing the Sales folder.

Notice in the figure above that 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 allows you to create web sites managed by thousands of people without centralized control. 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 a 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 using the checkboxes in the Security panel. 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 when developing a Zope application, you needn't perform any "manual" security checks. The term for this type of security which does not require manual effort on the part of the application developer is "declarative". Zope security is typically declarative. 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 from 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. We use Zope's `checkPermission` API to do this. 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 DTML we pass `this()` as the object which is a reference to the "current" object.

For Page Templates the syntax is a bit different, but the behavior is the same:

```
<form action="upload"
  tal:condition="python: modules['AccessControl'].getSecurityManager().checkPermission('Add Documents, Images, and Files',
  ...
  </form>
```

A Python Script can be employed to perform the same task on behalf of a Page Template. In the below example, we move the security check out of the Page Template and into a Python Script named `check_security`, which we call from the Page Template. Here is the Page template:

```
<form action="upload"
  tal:condition="python: here.check_security('Add Documents, Images and Files', here)">
```

Here is the `check_security` Python Script which is referenced within the Page Template:

```
## Script (Python) "check_security"
##bind container=container
##bind context=context
##bind namespace=
##bind script=script
##bind subpath=traverse_subpath
##parameters=permission, object
##title=Checks security on behalf of a caller

from AccessControl import getSecurityManager
sec_mgr = getSecurityManager()
return sec_mgr.checkPermission(permission, object)
```

You can see that permission checking may take place manually in any of Zope's logic objects. Other functions exist in the Zope API for manually performing security checks, but `checkPermission` is arguably the most useful.

By passing the current object to `checkPermission`, 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 object. 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 or the shortcut *user* in Page Templates. 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*.

You can do the same thing in a Page Template like this:

```
<p tal:content="user/getUserName">username</p>
```

The Zope security API for Scripts is explained in the Appendix B: API Reference . The Zope security API for DTML is explained in Appendix A: DTML Reference . The Zope security API for Page Templates is explained in Appendix C: Zope Page Templates Reference . An even better reference to these functions exists in the Zope help system, available by clicking on `Help` from any Zope Management Interface page.

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 computer systems are vulnerable to this style of attack. For web-based platforms, all that is required is to trick an authorized, but unsuspecting user to visit a URL that performs a harmful action that the attacker himself is not authorized to perform.

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 Trojan attacks by limiting the power of web resources based on who authored them. If an untrusted user authors 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 anyone attempt to view the page, it will fail since the owner of the object does 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, the user *owns* 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 the figure below.

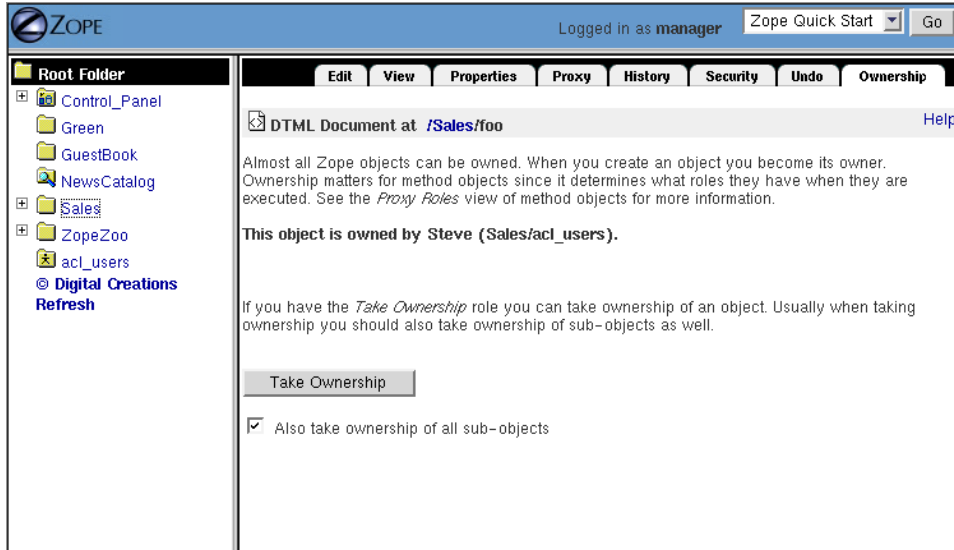


Figure 11-6 Managing ownership settings.

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 users will have the local role *Owner* on objects they own. However, ownership also affects security because it controls the role's executable content.

Note that due to the way Zope "grew up" that the list of users granted the *Owner* local role in the context of the object is *not* related to its actual "owner". The concepts of the owner "role" and executable content ownership are distinct. Just because someone has the *Owner* local role in the context of an executable object does not mean that he is the *owner* of the object.

Roles of Executable Content

DTML Documents, DTML Methods, SQL Methods, Python-based Scripts, and Perl-based Scripts are said to be *executable* since their content is generated dynamically. Their content is also editable through the web.

When you view an executable object by visiting its URL or calling it from DTML or a script, Zope runs the object's executable content. The object's actions are restricted by the roles of its 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 to protect against server-side Trojan horse attacks.

It is important to note that an "unowned" object is typically no longer executable. If you experience problems running an executable object, make sure that its ownership settings are correct.

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 its ownership 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 the figure below.

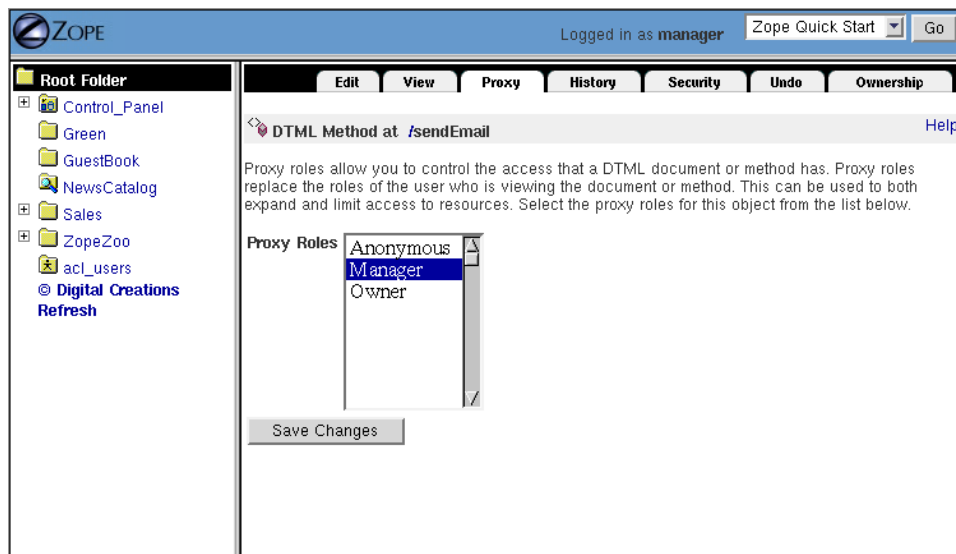


Figure 11-7 Proxy role management.

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 amount of permissions for 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 as having any other roles besides *Anonymous* despite the roles of the owner and viewer.

Use Proxy roles with care, since they can be used to 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.

Advanced DTML

DTML is the kind of language that appears to "do 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, well, it's no fun at all. This chapter tells you how to make DTML do what you *really* mean. When you're done reading this chapter you will be able to write DTML that will accomplish a number of complex tasks including:

- Inspect and Modify the REQUEST object
- Modify the current namespace
- Call other scripts from within DTML
- Send email with or without MIME attachments
- Handle exceptions within DTML

A few of caveats before getting started:

- It's a good idea to know something about Python before diving into advanced DTML or any other advanced area of Zope.
- Understand the Zope acquisition model and how it works.
- If you are writing very complex functionality in DTML, consider using a Python Script. This will ease maintenance, not to mention readability.
- Understand the difference between a DTML Document and a DTML Method before embarking on building a huge site. See the explanation included in this chapter.

It's no lie that DTML has a reputation for complexity. While it is true that DTML is really simple if all you want to do is simple layout, using DTML for more advanced tasks requires an understanding of where DTML variables come from.

Here's a very tricky error that almost all newbies encounter. Imagine you have a DTML Document 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 was 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 to 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 of sections, we'll show you how to fix this example to do what you mean.

How Variables are Looked up

There 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 knowing 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 be manipulated by *pushing* and *poping* 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 topmost object in the stack. If the name can't be found there, then the next item down is introspected. 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 the figure below.

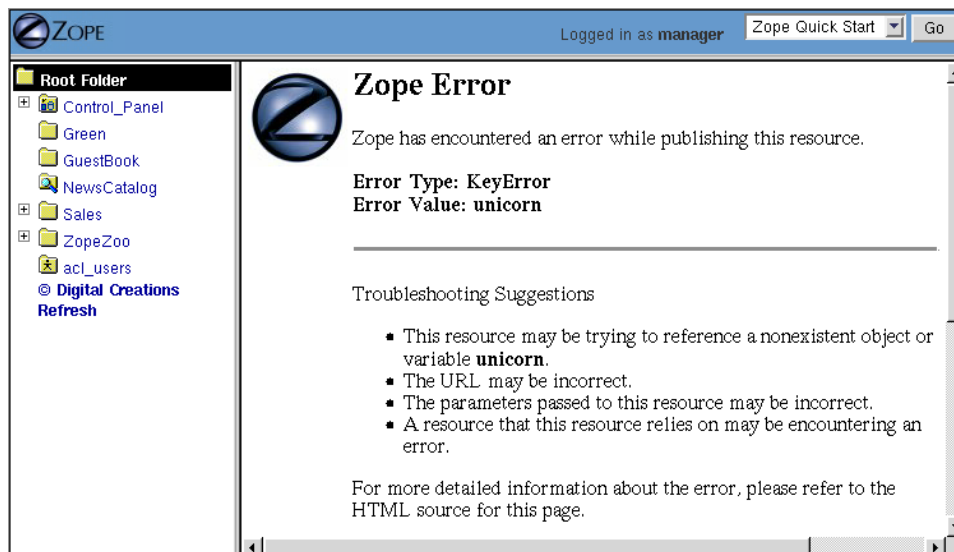


Figure 7-1 DTML error message indicating that it cannot find a variable.

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 the figure below.

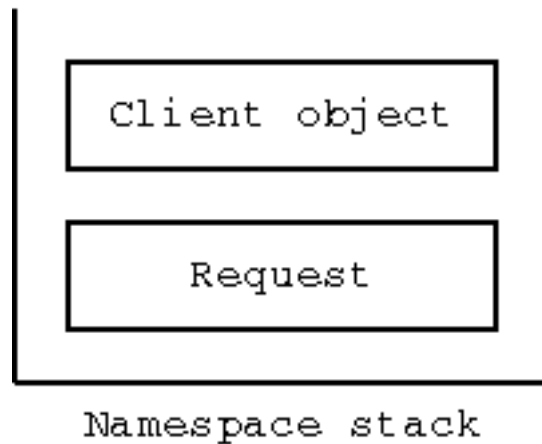


Figure 7-2 Initial DTML namespace stack.

The client object is the first object on the top of the DTML namespace stack when entering a transaction (note: commands exist to push additional parameters onto the namespace stack during a thread of execution). What the client object is depends on whether 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 the figure below.

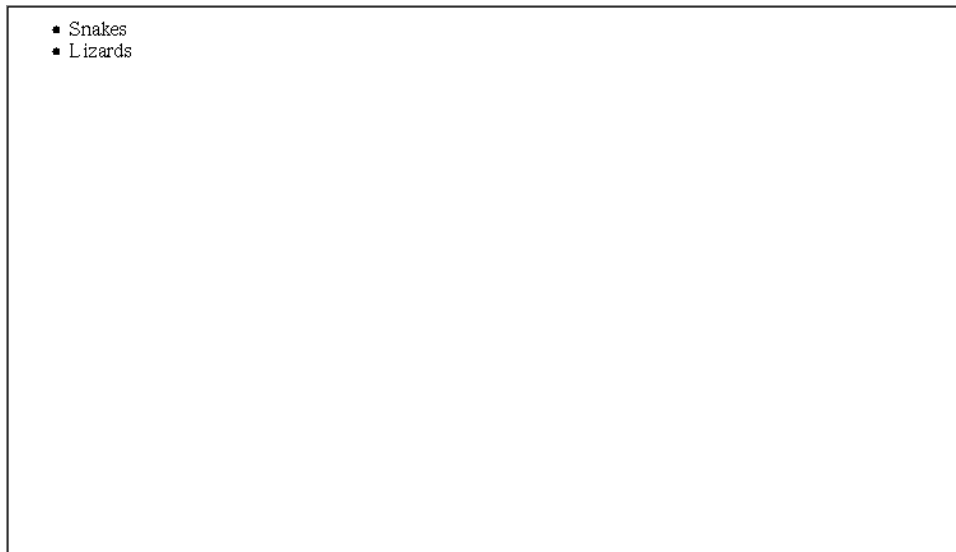


Figure 7-3 Applying the list method to the Reptiles folder.

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 "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 Method vs. DTML Document

One of the most potentially confusing choices to make for Zope newbies is the choice between a DTML Method and a DTML Document. Unfortunately, many Zope newbies develop entire sites using one type of object only to discover that they should have used the other type. In general, keep the following items in mind when deciding upon which type to use:

- **Does the object require properties of its own?** If so, use a DTML Document since DTML Methods have no inherent properties.
- **Does the object need to be called as a "page"?** If so, consider using a DTML Document since it will be easier to control such items as page title by using properties.
- **Does the object need transparency to its context?** If so, you should probably use a DTML Method since these objects act as though they are directly attached to their calling, or containing object.

DTML Request Object

The request object is the bottom 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 render the REQUEST object in a DTML Document or Method:

```
<dtml-var standard_html_header>
<dtml-var REQUEST>
<dtml-var standard_html_footer>
```

Try this yourself, you should get something that looks like the figure below.

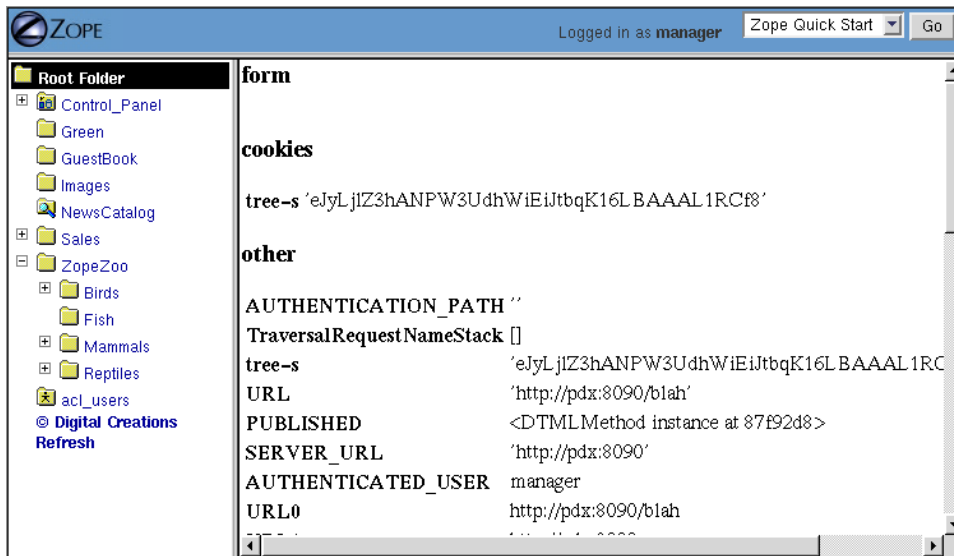


Figure 7-4 Displaying the request.

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 know 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.

Additional Notes

To be more accurate, the *in* tag pushes a number of items onto the namespace stack. These include sequence variables, grouping variables, and batch variables in addition to the object itself. Some of those variables are:

- `sequence-item`: The current item within the iteration.
- `sequence-start`: True if the current item is the first item in the sequence.
- `sequence-end`: True if the current item is the last item in the sequence.
- `sequence-length`: The length of the sequence.
- `previous-sequence`: True on the first iteration if the current batch is not the first one. Batch size is set with the `size` attribute.
- `next-sequence`: True on the last iteration if the current batch is not the last batch.

There are many more variables available when using the *in* tag. See Appendix A for more detail.

The With Tag

The *with* tag pushes an object that you specify onto 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 pushed onto the namespace stack:

```
<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 name-value pairs 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 the chapter entitled 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 alter the namespace in order to obtain references to the objects you need.

DTML Namespace Utility Functions

Like all things in Zope, the DTML namespace is an object, and it can be accessed directly in DTML with the `_` (underscore) object. The `_` namespace is often referred to 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(3,10,2)' -- 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 number of built-in names. The Zope philosophy 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. As mentioned above 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 square 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 text *selectedDoc* we want to insert the value of the variable named *selectedDoc*. For example, the value of *selectedDoc* might be `chapterOne`. Using this method, you can look up an item using a dynamic value instead of static text.

If you are 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 the chapter entitled 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, then the security system will raise an *Unauthorized* error and the user will be asked to present more privileged authentication credentials.

In the chapter entitled 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. Because the restrictions on looping and memory use are relatively tight, DTML is 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.

Another possibility for use of the *call* tag would be to call a ZSQL Method or or preprocess the REQUEST. Two examples of calling a ZSQL method:

```
<dtml-call "insertLogEntry(REQUEST)">
```

or:

```
<dtml-call "insertLogEntry(logInfo=REQUEST.get('URL0'), severity=1)">
```

To call a python script that might do any number of things, including preprocessing the REQUEST:

```
<dtml-call "preprocess(REQUEST)">
```

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.

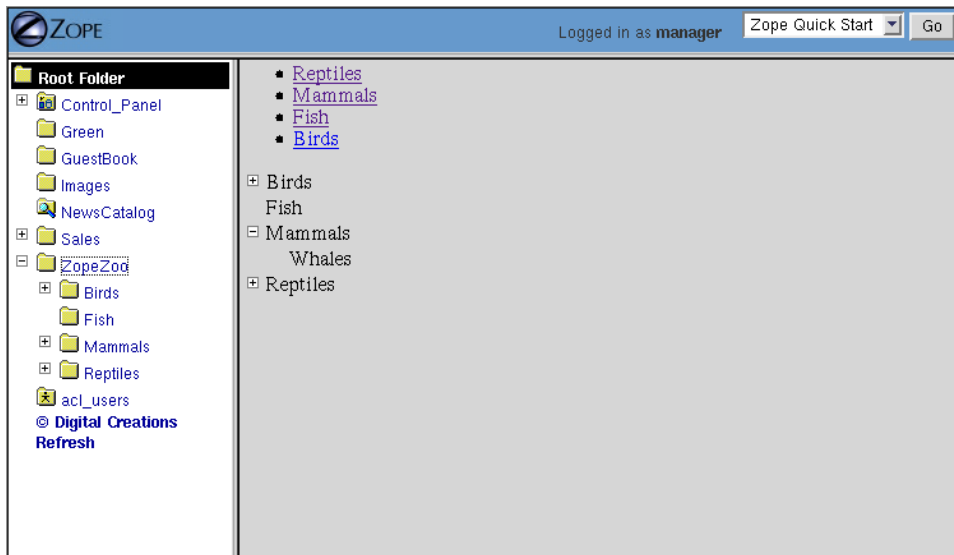


Figure 7-5 HTML tree generated by the tree tag.

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 default 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.

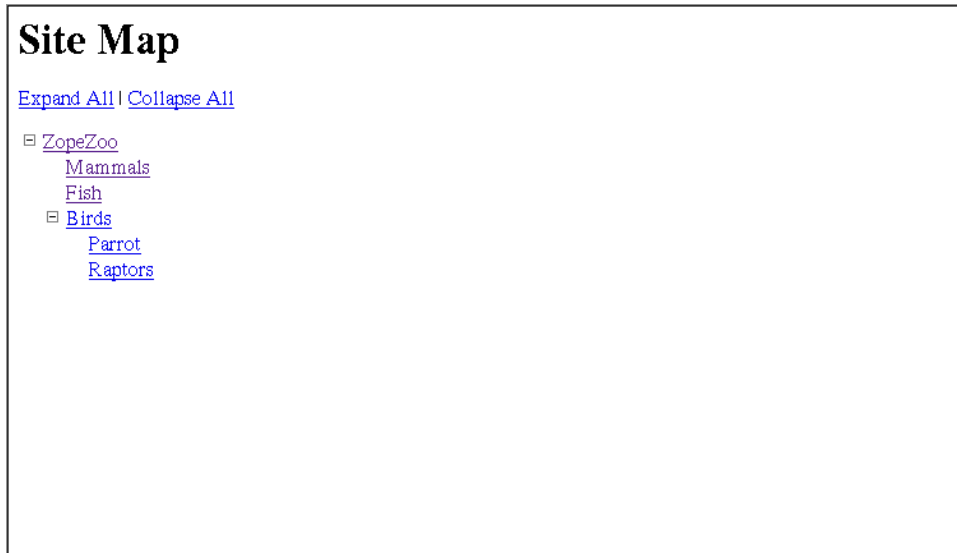


Figure 7-6 Dynamic site map using the tree tag.

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>
From: <dtml-var sender>
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>
```

Note: The text *:list* added to the name of the input fields directs Zope to treat the received information as a list type. For example if the first two checkboxes were selected in the above upload form, the REQUEST variable *send_to* would have the value [jobs@yahoo.com, jobs@microsoft.com]

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>
```

Another example utilizes the commonly accepted navigation scheme of presenting the the user page numbers from which to select:

```
<dtml-in "_range(1,101) "size=10 start=start">
  <dtml-if sequence-start>
    <p>Pages:
    <dtml-call "REQUEST.set('actual_page',1)">
    <dtml-in previous-batches mapping>
      <a href="<dtml-var URL><dtml-var sequence-query>start=<dtml-var "_['batch-start-index']+1">">
        <dtml-var sequence-number></a>&nbsp;
      <dtml-call "REQUEST.set('actual_page',_['sequence-number']+1)">
    </dtml-in>
    <b><dtml-var "_['actual_page']"></b>
  </dtml-if>
  <dtml-if sequence-end>
    <dtml-in next-batches mapping>&nbsp;
      <a href="<dtml-var URL><dtml-var sequence-query>start=<dtml-var "_['batch-start-index']+1">">
        <dtml-var "_['sequence-number']+_['actual_page']"></a>
    </dtml-in>
  </dtml-if>
</dtml-in>

<dtml-in "_range(1,101) "size=10 start=start">
  <br><dtml-var sequence-item>
</dtml-in>
```

This quick and easy method to display pages is a nice navigational tool for larger batches. It does present the drawback of having to utilize an additional *dtml-in* tag to iterate through the actual items, however.

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 the chapter entitled 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 are 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.

Other useful examples

In this section are several useful examples of dtml code. While many of these are most often better done in Python scripts, there are occasions when knowing how to accomplish this in dtml is worthwhile.

Forwarding a REQUEST

We have seen how to redirect the user's browser to another page with the help of the *call* directive. However, there are times when a redirection is not necessary and a simple forwarding of a REQUEST from one dtml-method to another would suffice. In this example, the dtml-method shown obtains a variable named *type* from the REQUEST object. A lookup table is reference to obtain the name of the dtml-method to which the REQUEST should be forwarded. The code below accomplishes this:

```
<dtml-let lookup="{ 'a' : 'form15', 'b' : 'form75', 'c' : 'form88' }">
  <dtml-return "_[lookup[REQUEST.get('type')]]">
</dtml-let>
```

This code looks up the name of the desired dtml-method in the lookup table (contained in the *let* statement) and in turn, looks up the name of this dtml-method in the current namespace. As long as the dtml-method exists, control will be passed to the method directly. This example could be made more complete with the addition of exception handling which was discussed above.

Sorting with the `<dtml-in>` tag

There are many times when sorting a result set is necessary. The *dtml-in* tag has some very interesting sort capabilities for both static and dynamic sorting. In the example below, a ZSQL method is called that returns results from a log table. The columns returned are logTime, logType, and userName. The dtml-method or document that contains this code will generate links back to itself to re-sort the query based upon certain search criteria:

```
<dtml-comment>

The sorting is accomplished by looking up a sort type
variable in the REQUEST that is comprised of two parts. All
but the last character indicate the name of the column on
which to sort. The last character of the sort type indicates
whether the sort should be ascending or descending.

</dtml-comment>

<table>
  <tr>
    <td>Time <a href="<dtml-var URL>?st=logTimea">A</a>&nbsp;<a href="<dtml-var URL>?st=logTimed">D</a></td>
    <td>Type <a href="<dtml-var URL>?st=logTypea">A</a>&nbsp;<a href="<dtml-var URL>?st=logTyped">D</a></td>
    <td>User <a href="<dtml-var URL>?st=userNamea">A</a>&nbsp;<a href="<dtml-var URL>?st=userNamed">D</a></td>
  </tr>

  <dtml-comment>The line below sets the default sort</dtml-comment>
  <dtml-if "REQUEST.get('st')==None"><dtml-call "REQUEST.set('st', 'logTimed')"></dtml-if>
  <dtml-in getLogData sort_expr="REQUEST.get('st')[0:-1]" reverse_expr="REQUEST.get('st')[-1]=='d'">
    <tr>
      <td><dtml-var logTime></td>
      <td><dtml-var logType></td>
      <td><dtml-var userName></td>
    </tr>
  </dtml-in>
</table>
```

Calling a DTML object from a Python Script

Although calling a DTML method from a Python script isn't really an advanced DTML technique, it deals with DTML, so it's being included here. To call a DTML Method or DTML Document from a Python script, the following code is used:

```
dtmlMethodName = 'index_html'  
return context[dtmlMethodName](container, container.REQUEST)
```

It's as simple as that. Often this is very useful if you wish to forward a request and significant processing is needed to determine which dtml object is the target.

Explicit Lookups

Occasionally it is useful to "turn off" acquisition when looking up an attribute. In this example, you have a folder which contains sub-folders. Each sub-folder contains Images. The top-level folder, each subfolder, and each image contain a property named *desc*.

If you were to query the Image for its *desc* property it would return the *desc* property of it's parent folder if the Image did not have the property. This could cause confusion as the Image would appear to have the *desc* property when it really belonged to the parent folder. In most cases, this behavior is desired. However, in this case, the user would like to see which images have the *desc* property and which don't. This is accomplished by utilizing *aq_explicit* in the call to the object in question.

Given the following structure:

```
Folder  
|  
|- Folder1 (desc='Folder one')  
|- Folder2 (desc='Folder two')  
    |  
    |- Image1 (desc='Photo one')  
    |- Image2  
    |- Image3 (desc='Photo three')
```

when the second image is asked for its *desc* property it will return `Folder two` based on acquisition rules:

```
<dtml-var "Image2.desc">
```

However, utilizing *aq_explicit* will cause Zope to look only in the desired location for the property:

```
<dtml-var "Image2.aq_explicit.desc">
```

This will, of course, raise an exception when the **desc** property does not exist. A safer way to do this is::

```
<dtml-if "_hasattr(Image2.aq_explicit, 'desc')">  
  <dtml-var "Image2.aq_explicit.desc">  
<dtml-else>  
  No desc property.  
</dtml-if>
```

As you can see, this can be very useful.

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.

Advanced Page Templates

In the chapter entitled Using Zope Page Templates you learned the basics features of Page Templates. In this chapter you'll learn about advanced techniques including new types of expressions and macros.

Advanced TAL

In this section we'll go over all TAL statements and their various options in depth. This material is covered more concisely in Appendix C, Zope Page Templates Reference .

In this chapter, the terms `tag` and `element` are used in the sense laid out by the XHTML spec . "<p>" is a *tag* , while the entire block "<p>stuff</p>" from opening tag through the closing tag is an *element* .

Advanced Content Insertion

You've already seen how `tal:content` and `tal:replace` work in the chapter entitled 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 convert HTML tags and entities in the text that they insert into an "escaped" form that appears in the resulting document as plain text rather than HTML markup. For instance, the '< character is "escaped" to `<`'. If you want to insert text as part of the HTML structure of your document, avoiding this conversion , you need to precede the expression with the `structure` keyword.

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' HTML into a series of paragraphs. The built-in variable `here` refers to the folder in which the template is rendered; See the "Expressions" section further below in this chapter for more information on `here` . In this case, we use `here` as the starting point for finding the Zope object `topNews` , which is presumably a list of news items or a Script which fetches such a list.

The `structure` keyword prevents the text of each `newsItem` value from being escaped. It doesn't matter whether the text actually contains any HTML markup, since `structure` really means "leave this text alone". This behavior is not the default because most of the text that you insert into a template will *not* contain HTML, but may contain characters that would interfere with the structure of your page.

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 to do anything special to make sure that dummy content is removed.

Default Content

You can leave the contents of an element 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 then its result will be inserted into the paragraph, otherwise it's Spam for dinner.

Advanced Repetition

You've already seen most of what you can do with the `tal:repeat` statement in the chapter entitled Using Zope Page Templates. This section covers a few advanced features of the `tal:repeat` statement.

Repeat Variables

One topic that bears more explanation are repeat variables. Repeat variables provide information about the current repetition. The following 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'`. The reason that you can't simply write `repeat/start` is that `tal:repeat` statements can be nested, so you need to be able to specify which one you want information about.

Repetition Tips

Here are a couple practical tips that you may find useful. Sometimes you'd like to repeat part of your template, but there is no naturally enclosing element. In this case, you must add an enclosing element, but you want to prevent it from appearing in the rendered page. You can do this with the `tal:omit-tag` statement:

```
<div tal:repeat="section here/getSections"
    tal:omit-tag="">
  <h4 tal:content="section/title">Title</h4>
  <p tal:content="section/text">quotation</p>
</div>
```

This is not just a matter of saving a few characters in the rendered output. Including the `div` tags in the output could affect the page layout, especially if it has stylesheets. We use the `tal omit-tag` statement to disinclude the `div` tag (and its pair closing tag) while leaving its contents unmolested. The `tal:omit-tag` statement is described in more detail 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)">
    <td tal:repeat="y python:range(1, 13)"
        tal:content="python:'%d x %d = %d' % (x, y, x*y)">
      X x Y = Z
    </td>
  </tr>
</table>
```

This example uses Python expressions, 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 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. 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 within an element on a tag and then use it many times within elements which are enclosed by this tag. 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 anywhere in the `ul` element. Notice also how you can have two TAL statements 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 it is false (in this case, an empty sequence) or true. If the `items` variable is false, then the `ul` element 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` element, then you can't use it in the `ul` element any more, because it becomes a *local* variable of the `h4` element. You could place the definition on some element 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 `span` tag to the bottom of the template:

```
<span tal:define="global items container/objectIds"></span>
<h4 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;
             global untitled not:title;
             tlen python:len(title);">
```

In this case, both `title` and `tlen` are local to the paragraph, but `untitled` is global. 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 opening and closing tags, but does not affect the contents of the element. 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` can also be used with a true/false expression, in which case it only removes the tags if the expression is true. For example:

```
Friends: <span tal:repeat="friend friends">
  <b tal:omit-tag="not:friend/best"
     tal:content="friend/name">Fred</b>
</span>
```

This will produce a list of friends, with our "best" friend's name in bold.

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 element, then on its enclosing element, and so on until it reaches the top-level element. When it finds an error handler, it replaces the contents of that element with the error handling expression. In this case, the `span` element will contain an error message.

Typically you'll define an error handler on an element 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 it. 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 are 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 statements are likely to depend on it and since it should come after `define` and `repeat`.

Here's an example element 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 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 due to TAL statement execution order. Despite the order in which they are written, the condition is always tested before the repeat is executed. This results in a situation in which the `n` variable is not defined until after it is tested, which ultimately causes an error when you attempt to test or otherwise view the template. 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` element. Notice that the `div` tag will not appear in the output due to its `tal:omit-tag` statement.

Although `span` and `div` are natural choices for this in HTML, there is, in general, no equivalent natural element in XML. In this case, you can use TAL's namespace in a new way: while TAL does not define any tags, it doesn't prohibit any either. You can make up any tag name you like within the TAL namespace, and use it to make an element, like so:

```
<tal:series define="items here/getItems">
  <tal:items repeat="item items">
    <tal:parts repeat="part item">
      <part tal:content="part">Part</part>
    </tal:parts>
  </tal:items>
  <noparts tal:condition="not:items" />
</tal:series>
```

The `tal:series`, `tal:items`, and `tal:parts` tags in this example should be acceptable to tools that handle XML namespaces properly, and to many HTML tools. This method has two additional advantages over a `div`. First, TAL tags are omitted just like TAL attributes, so no `tal:omit-tag` is necessary. Second, TAL attributes in these tags don't require their own `tal:` prefix, since they inherit the namespace of the tag. The METAL namespace can be used in exactly the same 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 documents: 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 the chapter entitled *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 calls 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 input 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, or whatever it needs to do to "get the job done". Here's a sketch of how to validate input with a 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. The Script's `context` variable is equivalent to `here` in TALES. You can pass Page Templates extra information with keyword arguments. The keyword arguments are available to the template via the `options` built-in variable. So the form template in this example might include a section like this:

```
<span tal:condition="options/error_message | nothing">
Error: <b tal:content="options/error_message">
    Error message goes here.
</b></span>
```

This example shows how you can display an error message that is passed to the template via keyword arguments. Notice the use of `| nothing` to handle the case where no `error_message` argument has been passed to the template.

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 need to set up a quick-and-dirty form, you can always create a version of the form-action pair using Page Templates alone. You should only do this when you don't care about error handling and when the response will always be the same, no 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, in the TAL statement `<td tal:content="request/form/age">Age</td>`, the expression of the statement is `request/form/age`. `request/form/age` is an example of a *path expression*. Path expressions describe objects by giving them paths such as `request/form/age`, or `user/getUserName`. Expressions only work in the context of a TAL statement; they do not work in "normal" HTML inserted in your page templates. In this section you'll learn about all the different types of expressions, and variables.

Built-in Page Template 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. Note that these variables are different than the built-in variables that you would use in a Script (Python), they are only effective for Page Templates::

nothing — A false value, similar to a blank string, that you can use in `tal:replace` or `tal:content` to erase an element or its contents. If you set an attribute to `nothing` , the attribute is removed from the tag (or not inserted). A blank string, on the other hand, would insert the tag with an empty value, as in `alt=""` .

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. When a template is rendered from the web, no options are present. Options are only available when a template is called from Python or by similarly complex means. For example, when the template `t` is called by the Python expression `t(foo=1)` , the path `options/foo` equals `1` .

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 throughout 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 Fred Flintstone."
```

If the path expression has more than one part (if it contains a slash), 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 `"` . 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 the chapter entitled 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 the chapter entitled 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, zero, empty strings, empty sequences, nothing, and None are considered false, while everything else is true. Non-existent paths are neither true nor false, and applying a `not:` to such a path will fail.

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 the 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 the chapter entitled 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 is no support in TAL to do this without Python expressions. In 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, printing information about widgets which are of type `gear` .

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>
```

If the user is `Anonymous` , then the `span` element is replaced with the text `need to log in` . Otherwise, the default content is used, which is in this case `are logged in as ...` .

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')">
```

This assigns `oddclass` and `evenclass` class attributes to alternate rows of the table, allowing them to be styled differently in HTML output, for example.

Without the `test` function you'd have to write two `tr` elements with different conditions, 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 exist *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 access the `penguin.gif` object with the following Python expression:

```
"python:here.penguin.gif"
```

Instead, 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 the chapter entitled Variables and Advanced DTML for more information on explicitly calling DTML.

When 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. For example, here is a template that uses a DTML Method to generate JavaScript:

```
<head tal:define="items here/getItems.sql">
  <title tal:content="template/title">Title</title>
  <script tal:content="structure here/jsItems"></script>
</head>
...etc...
```

...and here is the DTML Method 'jsItems':

```
<dtml-let prefix="template.id">
<dtml-in items>
&dtml-prefix;_&dtml-name; = &dtml-value; ;
</dtml-in>
</dtml-let>
```

The DTML uses the template's `id` , and the `items` variable that it defined just before the call.

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']"
```

Many Python modules cannot be accessed from Page Templates, DTML, or Scripts unless you add Zope security assertions to them. See the Zope Developer's Guide's security chapter for more information on making more Python modules available to your templates and scripts by using "ModuleSecurityInfo".

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, much as you can with DTML (e.g. by inserting `standard_html_header` and `standard_html_footer` into your DTML), but in a slightly different way.

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` element (including all contained elements).

Macros defined in a Page Template are stored in the template's `macros` 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` element 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) 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 to using them which are worth mentioning.

A macro's name must be unique within the Page Template in which it is 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>
```

In this case the path expression returns a macro defined dynamically by the `getMacro` script. 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">
```


The Zope Book (2.6 Edition)

```
This content remains - no macro is used
</p>
```

The result is the same as using `default` with `tal:content` and `tal:replace`. The "default" content in the tag doesn't change when it is rendered. This can be handy if you need to conditionally use a macro or fall back on the default content if it doesn't exist.

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. When you're editing in the ZMI, rather than using a WYSIWYG editing tool, it's more convenient not to expand macros when editing. This is the default for newly created templates. When using WYSIWYG tools, however, it is often desirable to have the macros expanded so you are editing a complete page. In this case, check the *Expand macros..* checkbox before editing the page.

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:

```
<div 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>
</div>
```

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:

```
<div 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>
</div>
```

When you use this macro you can choose to fill the slot like so:

```
<p metal:use-macro="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:

```
<div>
  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>
</div>
```

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">
  <div 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>
  </div>
  <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">
  <div 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>
  </div>
  <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="href link/url"
      tal:content="link/name">link name</a>
  </li>
</ul>
```

In this case, `getLinks` is an (imaginary) Script that assembles a list of link objects, possibly using a Catalog query.

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 by redefining the `getLinks` Script at different locations within your site.

It's not always easy to figure out the best way to 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 than 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 pages 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 the chapter entitled 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 your Page Template and selecting the cache manager (there must be one in the acquisition path of the template for the *Cache* view to appear), 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.

For more information on caching in the context of Zope, see the chapter entitled *Zope Services*.

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 hundreds of 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-blown 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:int(path('request/start | nothing') or 0);
    batch python:modules['ZTUtils'].Batch(employees,
                                          size=3,
                                          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>
```

Define a Script (Python) with the name `getEmployees` in the same folder with the following body (no parameters are necessary):

```
return [ {'name': 'Chris McDonough', 'salary': '5'},
         {'name': 'Guido van Rossum', 'salary': '10'},
         {'name': 'Casey Duncan', 'salary': '20'},
         {'name': 'Andrew Sawyers', 'salary': '30'},
         {'name': 'Evan Simpson', 'salary': '35'},
         {'name': 'Stephanie Hand', 'salary': '40'}, ]
```

This example iterates over batches of results from the `getEmployees` 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. The batch size in this case is 3.

Take a look at the `tal:define` statement on the `body` element. It defines a bunch of batching variables. The `employees` variable is a list of employee objects returned by the `getEmployees` Script. It is not very big now, but it could grow fairly large (especially if it were a call into a SQL Method of *real* employees). 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 no link. The `tal:attributes` statement creates a link to the previous and next batches. The link is simply the URL of 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 starting 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.

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.

As mentioned earlier in the chapter, the `sequence` module provides a handy `sort` function.

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 on these utilities.

Conclusion

This chapter covers some useful and some obscure 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 if and when you need them.

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 Page Templates, and data with files and images. This chapter shows you how to add *Script* objects which allow you to write scripts in Python and Perl through your web browser.

What is *logic* and how does it differ from presentation? Logic provides those actions which change objects, send messages, test conditions and respond to events, whereas presentation formats and displays information and reports. Typically you will use DTML or Page Templates 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. Script objects first appeared in Zope 2.3, and are now the preferred way to write programming logic in Zope. Currently, Zope comes with *Python-based Scripts*, which are written in the Python language. There is a third-party extension to Zope, available as a separate download, which allows you to write *Perl-based Scripts* in the Perl language.

So far in this book you have heavily used DTML Methods, DTML Documents, and Page Templates (ZPT) to create simple web applications in Zope. DTML and ZPT allow you to perform simple scripting operations such as string manipulation. For the most part, however, DTML and ZPT should be used for presentation. DTML Methods are explained in the chapter entitled Basic DTML, and the chapter entitled Advanced DTML. ZPT is explained in the chapter entitled Using Zope Page Templates, and the chapter entitled Advanced Page Templates.

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.

External Methods — These are also written in Python, but the code is stored on the filesystem. External Methods allow you to do many things that are restricted from Python-based Scripts for security reasons.

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. Currently you must download and install third-party extensions before you can use Perl scripts in Zope.

You can add these scripts to your Zope application just like any other object. Details about each type of script are provided below, in the sections "Using Python-based Scripts," "Using External Methods," and "Using Perl-based Scripts" respectively.

Calling Scripts

Any Zope script may be called "from the web" (for example, from a browser or another web-aware tool). In addition, any type of script may 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 the chapter entitled Basic 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.

Context

When you call a script, you usually want to single out some object that is central to the script's task, either because that object provides information that the script needs, or because the script will modify that object. In object-oriented terms, we want to call the script as a *method* of this particular object. But in conventional object-oriented programming, each object can perform the methods that are defined in (or inherited by) its class. How is it that one Zope script can be used as a method of potentially many objects, without the script being defined in the classes that define these objects?

Recall that in the chapter entitled Acquisition, we learned that Zope can find objects in different places by *acquiring* them from parent containers. Acquisition allows us to treat a script as a method that can be called *in the context* of any suitable object, just by constructing an appropriate URL. The object on which we call a script gives it a context in which to execute. It is simpler to just say that you are calling the script *on* the object. Or, to put it another way, **the context is the environment in which the Script executes**, from which the script may get information that it needs to do its job.

Another way to understand the context of a script is to think of the script as a function in a procedural programming language, and its context as an implicit argument to that function.

There are two general ways to call a script and provide it with a context: by visiting a URL, and by calling the script from another script or template.

Calling Scripts From the Web

You can call a script directly with a web browser by visiting its URL. You can call a single script on different objects by using different URLs. This works because Zope can determine the script's context by URL. 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 the figure below.

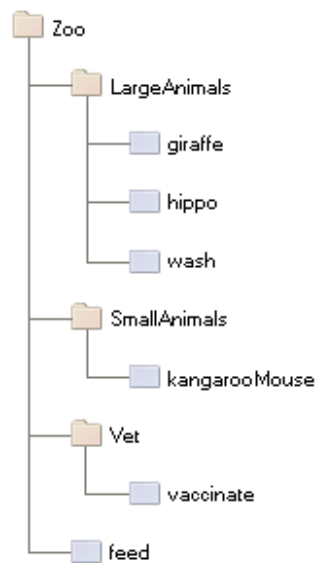


Figure 14-1 A collection of objects and scripts

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 *kangarooMouse* objects, respectively.

Zope uses a URL as a map to find which object and which script you want to call.

URL Traversal and Acquisition

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 cannot be found in the *hippo* object and is located in the *Zoo* folder by using acquisition. Zope always starts looking for an object in the last object it traversed, in this case *hippo* . Since *hippo* does not contain anything, Zope backs up to *hippo*'s immediate container, *LargeAnimals* . The *feed* script is not there, so Zope backs up to *LargeAnimals*' container, *Zoo* , where *feed* is finally found.

Now Zope has reached the end of the URL and has matched objects to every name in the URL. Zope recognizes that the last object found, *feed* , is callable, and calls it in the context of 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* by visiting the URL *Zoo/LargeAnimals/hippo/wash* . In this case Zope acquires the *wash* method from the *LargeAnimals* folder.

Passing Arguments with an HTTP Query String

You can pass arguments to a URL, too. Just append them as standard query strings:

```
http://my-zope-site:8080/Zoo/LargeAnimals/hippo/wash?soap=lye
```

Calling Scripts from Other Objects

You can call scripts from other objects, whether they are DTML objects, Page Templates, or Scripts (Python or Perl). The semantics of each language differ slightly, but the same rules of acquisition apply. You do not necessarily have to know what language is used in the script you are calling; you only need to pass it any parameters that it requires, if any.

Calling Scripts from DTML

As you saw in the chapter entitled *Advanced DTML* , you can call Zope scripts from DTML with the *call* tag. For example:

```
<dtml-call updateInfo>
```

DTML will call the *updateInfo* script, whether it is implemented in Perl, Python, or any other language. You can also call other DTML objects and SQL Methods the same way.

If the *updateInfo* script requires parameters, either your script must have a name for the DTML namespace binding (see *Binding Variables* in the section "Using Python-based Scripts" below), so that the parameters will be looked up in the DTML namespace, or you must pass the parameters in an expression. For example::

```
<dtml-call expr="updateInfo(color='brown', pattern='spotted')">
```

You can also pass in any variables that are valid in the current DTML namespace. For example, if *newColor* and *newPattern* are defined using *dtml-let*, you could pass the variables as parameters like this:

```
<dtml-call expr="updateInfo(color=newColor, pattern=newPattern)">
```

You can also pass variables that are defined automatically by dtml tags such as *dtml-in*. For example:

```
<dtml-in all_animals prefix="seq">
  <dtml-call expr="feed(animal=seq_item)">
</dtml-in>
```

This assumes that *feed* is a script and has a parameter called *animal*. The standard names used during DTML loops (sequence-item, sequence-key, et al.) are a bit cumbersome to spell out in a Python *expr*, because "sequence-item" would be interpreted as *sequence* minus *item*. To avoid this problem, we use the *prefix* attribute of the dtml-in tag, which uses the specified value ("seq") and an underscore ("_") instead of the customary "sequence-" string.

Calling scripts from Python and Perl

Calling scripts from other Python or Perl scripts works the same as calling scripts from DTML, except that you must *always* pass script parameters when you call a script from Python or Perl. For example, here is how you might call the *updateInfo* script from Python:

```
new_color='brown'
context.updateInfo(color=new_color,
                  pattern="spotted")
```

Note the use of the *context* variable to tell Zope to find *updateInfo* by acquisition.

From Perl you could do the same thing using standard Perl semantics for calling scripts:

```
$new_color = 'brown';
$self->updateInfo(color => $new_color,
                 pattern => "spotted");
```

Here we see that *self* is the way we refer to the current context in a Perl-based script.

Zope locates the scripts you call by 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. The figure below shows a slightly updated object hierarchy that contains two scripts, *vaccinateHippo.py* and *vaccinateHippo.pl*.

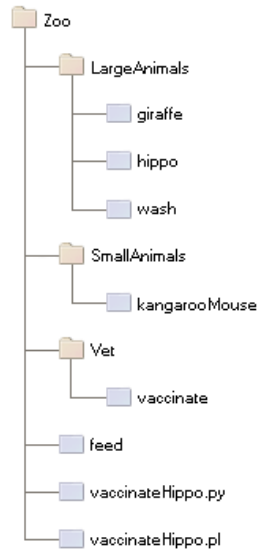


Figure 14-2 A collection of objects and scripts

Suppose *vaccinateHippo.py* is a Python script. Here is how you can call the *vaccinate* script on the *hippo* object from the *vaccinateHippo.py* script:

```
context.Vet.LargeAnimals.hippo.vaccinate()
```

In other words, you simply access the object by using the same acquisition path as you would use if you called it from the web. The result is the same as if you visited the URL *Zoo/Vet/LargeAnimals/hippo/vaccinate*. Note that in this Python example, we do not bother to specify *Zoo* before *Vet*. We can leave *Zoo* out because all of the objects involved, including the script, are in the *Zoo* folder, so it is implicitly part of the acquisition chain.

Likewise, in the Perl version, *vaccinateHippo.pl*, you could say:

```
$self->Vet->LargeAnimals->hippo->vaccinate();
```

Calling Scripts from Page Templates

Calling scripts from Page Templates is much like calling them by URL or from Python. Just use standard TALES path expressions as described in the chapter entitled Using Zope Page Templates. For example:

```
<div tal:replace="here/hippo/feed" />
```

The inserted value will be HTML-quoted. You can disable quoting by using the *structure* keyword, as described in the chapter entitled Advanced Page Templates.

Page Templates do not really provide an equivalent to DTML's *call* tag. To call a script without inserting a value in the page, you can use *define* and ignore the variable assigned:

```
<div tal:define="dummy here/hippo/feed" />
```

In a page template, *here* refers to the current context. It behaves much like the *context* variable in a Python-based Script. In other words, *hippo* and *feed* will both be looked up by acquisition.

If the script you call requires arguments, you must use a TALES python expression in your template, like so:

```
<div tal:replace="python:here.hippo.feed(food='spam') " />
```

Just as in Path Expressions, the `here` variable refers to the acquisition context the Page Template is called in.

The python expression above is exactly like a line of code you might write in a Script (Python). The only difference is the name of the variable used to get the acquisition context. Don't be misled by the different terminology: context is context, whatever you call it. Unfortunately, the different names used in ZPT and Python Scripts evolved independently. (Note that as of this writing, the ZPT variable `here` is planned to become `context` in a future version of Zope, probably Zope 3.)

For further reading on using Scripts in Page Templates, refer to the chapter entitled Advanced Page Templates .

Calling Scripts: Summary and Comparison

Let's recap the ways to call a hypothetical `updateInfo` script on a `foo` object, with argument passing: from your web browser, from Python, from Perl, from DTML, and from Page Templates.

by URL:

```
http://my-zope-server.com:8080/foo/updateInfo?amount=lots
```

from a Python script:

```
context.foo.updateInfo(amount="lots")
```

from a Perl script:

```
$self->foo->updateInfo(amount="lots");
```

from a Page Template:

```
<span tal:content="dummy here/foo/updateInfo"/>
```

from a Page Template, with arguments:

```
<span tal:content="python:here.foo.updateInfo(amount='lots')"/>
```

from DTML:

```
<dtml-with foo >
  <dtml-var updateInfo>
</dtml-with>
```

from DTML, with arguments:

```
<dtml-with foo>
  <dtml-var expr="updateInfo(amount='lots')">
</dtml-with>
```

another DTML variant:

```
<dtml-var expr="_['foo'].updateInfo()>
```

Regardless of the language used, this is a very common idiom to find an object, be it a script or any other kind of object: you ask the context for it, and if it exists in this context or can be acquired from it, it will be used.

Zope will throw a `KeyError` exception if the script you are calling cannot be acquired. If you are not certain that a given script exists in the current context, or if you want to compute the script name at run-time, you can use this Python idiom:

```
updateInfo = getattr(context, "updateInfo", None)
if updateInfo is not None:
    updateInfo(color="brown", pattern="spotted")
```

```
else:  
    # complain about missing script
```

The `getattr` function is a Python built-in. The first argument specifies an object, the second an attribute name. The `getattr` function will return the named attribute, or the third argument if the attribute cannot be found. So in the next statement we just have to test whether the `updateInfo` variable is `None`, and if not, we know we can call it.

Using Python-based Scripts

Earlier in this chapter you saw some examples of scripts. Now let us 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.

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. Our script is now equivalent to the following function definition in standard Python syntax:

```
def hello(name="World"):  
    return "Hello %s." % name
```

The result should appear something like the below image:

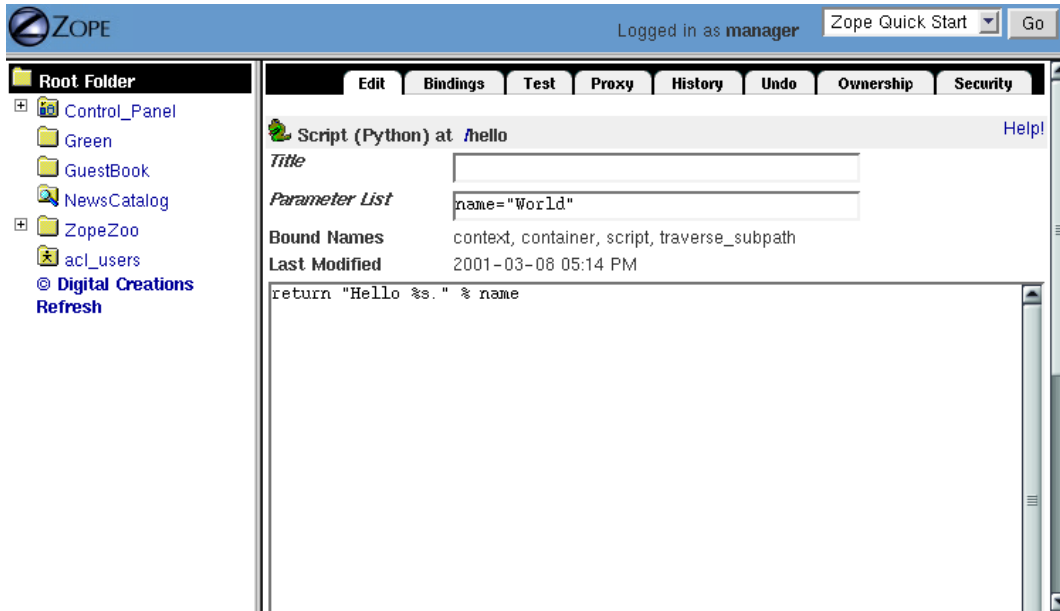


Figure 14-3 Script editing view

You can now test the script by going to the *Test* tab as shown in the figure below.

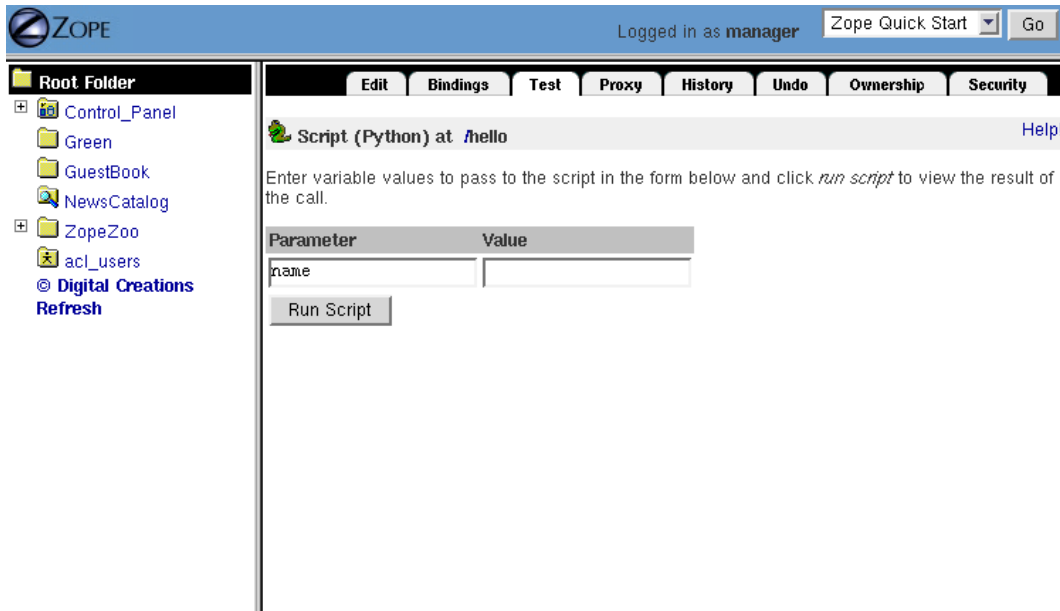


Figure 14-4 Testing a Script

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, as described above in the section "Calling Scripts". 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()`, a method in the Zope API, to get a list of the contained objects. `objectIds` is a method of `Folders`, so the `context` object should be a `Folder`-like object. The script then calls `len()` to find the number of items in that list. 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. The lines at the top starting with a double hash (`##`) are generated by Zope when editing a script via FTP. You can specify parameters and other things here (this is covered in more detail in the next section, *Binding Variables*).

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 (or Page Templates) for presentation or report formatting, while Python handles the logic. This is a very important pattern that you will witness and use repeatedly in Zope.

Binding Variables

A set of special variables is created whenever a Python-based Script is called. These variables, defined on the script's *Bindings* view in the Zope Management Interface, are used by your script to access other Zope objects and scripts. They are not available in other Zope objects such as Perl scripts or DTML Documents, though there is a similar set of variables available in ZPT.

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 in which the script is defined.

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. 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 the chapter entitled Advanced DTML . Furthermore, 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 will 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. Note the implication that these comments are not merely comments: they carry semantic significance.

What are all these bindings good for, anyway? They can be used to control where Zope looks for the objects you need. We have been explaining the idea of *context* throughout this chapter; not surprisingly, in a Python-based Script, you can access its context through the *context* binding. Returning to our Zoo example, our *feed* script might contain a line like this:

```
animal_id = context.getId()
```

In this example, we get the value "hippo," because *hippo* is the context and its id is "hippo". But we can call `getId()` on other variables, since nearly all Zope objects support this method. For example:

```
folder_id = container.getId()
script_id = script.getId()
```

These values depend on the id of the script and the container it lives in. Calling the script in a different context will have no effect on these variables. Acquisition still applies when using *container* , but Zope will only try to acquire from the script's containers, not from the calling context.

Accessing the HTTP Request

What if we need to get user input, e.g. values from a form? We can find the REQUEST object, which represents a Zope web request, in the context. For example, if we visited our *feed* script via the URL `Zoo/LargeAnimals/hippo/feed?food_type=spam`, we could access the *food_type* variable as `context.REQUEST.food_type` . This same technique works with variables passed from forms.

Another way to get the REQUEST is to pass it as a parameter to the script. If REQUEST is one of the script's parameters, Zope will automatically pass the HTTP request and assign it to this parameter. We could then access the `food_type` variable as `REQUEST.food_type`.

String Processing in Python

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 within Python-based Scripts because of security restrictions. If you really need regular expressions, you can easily use them from External Methods, described in a subsequent section of this chapter. However, in a Script (Python), 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. Python 2.X also provides "string methods" which can perform most of the same duties as the `string` module.

Suppose you want to change all the occurrences of a given word in a DTML Document. Here is 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 will need permission to edit a document in order
to call this script on the document.
This script assumes that the context is a DTML document,
which provides the document_src() and manage_edit() methods
described in Appendix B (API Reference).
"""
import string
text=context.document_src()
text=string.replace(text, word, replacement)
context.manage_edit(text, context.title)
```

You can perform the same job by using the Python string method 'replace':

```
## Script (Python) "replaceWord"
##parameters=word, replacement
##
text=context.document_src()
text=text.replace(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=Alligator&replacement=Crocodile` would call the `replaceWord` script on the document named `Swamp` and would replace all occurrences of the word `Alligator` with `Crocodile`.

You could also call this script from a DTML method, from a Page Template, or even from another Script, as described in this chapter under the heading "Calling Scripts from other Objects."

See the Python documentation for more information about manipulating strings from Python.

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 the chapter entitled 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 or ZPT. 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 is 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 an HTML *IMG* tag that references a random image in the *Images* Folder.

A ZPT equivalent to the above DTML script is:

```
<span tal:replace="here/Images/randomImage" />
```

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. And for control of presentation, you should return the script output to a Page Template or DTML page which then displays the return value appropriately.

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.

The following 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 fewer restrictions than Python-based Scripts.

Using External Methods

Sometimes the security constraints imposed by scripts, DTML and ZPT 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 can use *External Methods* . We encountered External Methods briefly in the chapter entitled Using Basic Zope Objects . Now we will explore them in more detail.

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 server's 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.

External Method code is 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 your External Methods in an *Extensions* directory inside an installed Zope product directory, or in your `INSTANCE_HOME` directory if you have one. See the chapter entitled Installing and Starting Zope for more about `INSTANCE_HOME`.

Let's take an example. Create a file named *Example.py* in the *Zope Extensions* directory on your server. In the file, enter the following code:

```
def hello(name="World"):
    return "Hello %s." % name
```

You've created a Python function in a Python module. But you have not yet created an External Method from it. To do so, we must add an External Method object in Zope.

To add an External Method, choose *External Method* from the product add list. You will be taken to a form where you must provide an id. Type "hello" into the *Id* field, type "hello" in the *Function name* field, and type "Example" in the *Module name* field. Then 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 the figure below.

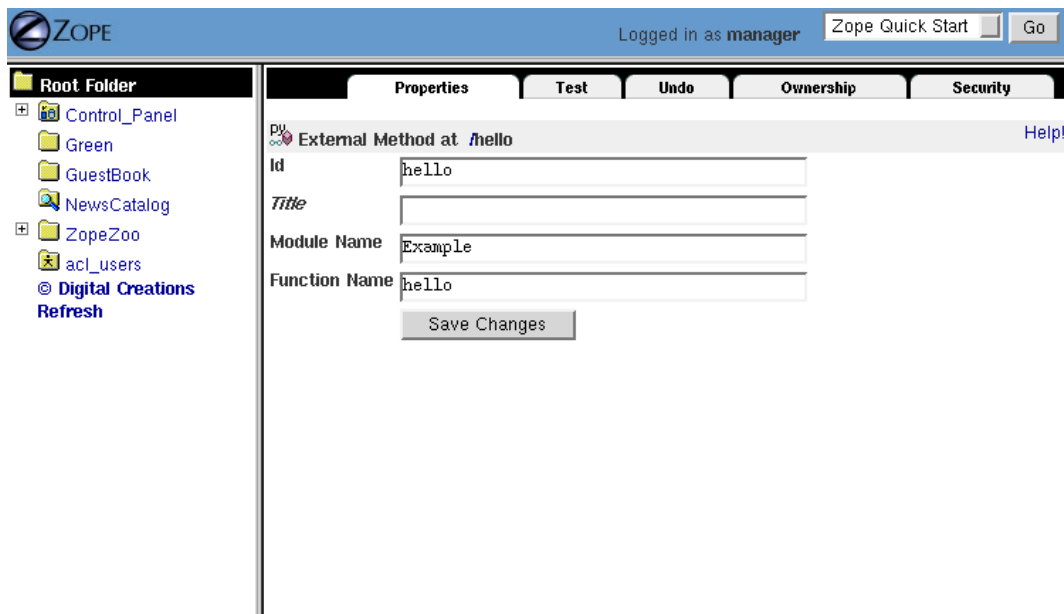


Figure 14-5 External Method Properties view

Note that if you wish to create several related External Methods, you do not need to create multiple modules on the filesystem. You can define any number of functions in one module, and add an External Method to Zope for each function. For each of these External Methods, the *module name* would be the same, but *function name* would vary.

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 Python-based scripts. In fact, for simple string processing tasks like this, scripts offer a better solution since they are easier to work with.

The main reasons to use an External Method are to access the filesystem or network, or to use Python packages that are not available to restricted scripts.

For example, a Script (Python) cannot access environment variables on the host system. One could access them using an External Method, like so:

```
def instance_home():
    import os
    return os.environ.get('INSTANCE_HOME')
```

Regular expressions are another useful tool that are restricted from Scripts. Let's look at an example. Assume we want to get the body of an HTML Page (everything between the `body` and `/body` tags):

```
import re
pattern = r"<\s*body.*?>(.*?)</body>"
regexp = re.compile(pattern, re.IGNORECASE + re.DOTALL)

def extract_body(htmlstring):
    """
    If htmlstring is a complete HTML page, return the string
    between (the first) <body> ... </body> tags
    """
    matched = regexp.search(htmlpage)
    if matched is None: return "No match found"
    body = matched.group(1)
    return body
```

Note that we import the `re` module and define the regular expression at the module level, instead of in the function itself; the `extract_body()` function will find it anyway. Thus, the regular expression is compiled once, when Zope first loads the External Method, rather than every time this External Method is called. This is a common optimization tactic.

Now put this code in a module called `my_extensions.py`. Add an External Method with an id of `'body_external_m'`; specify `my_extensions` for the Module Name to use and, `extract_body` for Function Name.

You could call this for example in a Script (Python) called `store_html` like this:

```
## Script (Python) "store_html"
##

# code to get 'htmlpage' goes here...
htmlpage = "some string, perhaps from an uploaded file"
# now extract the body
body = context.body_external_m(htmlpage)
# now do something with 'body' ...
```

... assuming that `body_external_m` can be acquired by `store_html`. This is obviously not a complete example; you would want to get a real HTML page instead of a hardcoded one, and you would do something sensible with the value returned by your External Method.

Here is 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 an
    'original_id' property set to the id of the full size image
    object.
    """

    import PIL
    from StringIO import StringIO
    import os.path
    # none of the above imports would be allowed in Script (Python)!

    # Note that PIL.Image objects expect to get and save data
    # from the filesystem; so do Zope Images. We can get around
    # this and do everything in memory by using StringIO.
    # Get the original image data in memory.
```

```
original_image=getattr(self, original_id)
original_file=StringIO(str(original_image.data))

# create the thumbnail data in a new PIL Image.
image=PIL.Image.open(original_file)
image=image.convert('RGB')
image.thumbnail((size,size))

# get the thumbnail data in memory.
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 an old thumbnail, delete it
if thumbnail_id in self.objectIds():
    self.manage_delObjects([thumbnail_id])

# create the Zope image object for the new thumbnail
self.manage_addProduct['OFSP'].manage_addImage(thumbnail_id,
                                                thumbnail_file,
                                                'thumbnail image')

# now find the new zope object so we can modify
# its properties.
thumbnail_image=getattr(self, thumbnail_id)
thumbnail_image.manage_addProperty('original_id', original_id, 'string')
```

Notice that the first parameter to the above function is called *self*. This parameter is optional. If *self* is the first parameter to an External Method function definition, it will be assigned the value of the calling context (in this case, a folder). It can be used much like the *context* we have seen in Scripts (Python).

You must have PIL installed for this example to work. Installing PIL is beyond the scope of this book, but note that it is important to choose a version of PIL that is compatible with the version of Python that is used by your version of Zope. See the PythonWorks website for more information on PIL.

To continue our example, 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 (Python) 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 not 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 the figure below.

Thumbnails



Figure 14-6 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 a good way 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 such as PIL while your scripts do simple processing of Zope objects. Note that you could just as easily use a Page Template instead of DTML.

Processing XML with External Methods

You can use External Methods to do nearly 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 you 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 your External Method and assign an instance of this class 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.

If you need to create new kinds of persistent objects, it's time to learn about writing Zope Products. Writing a Product is beyond the scope of this book. You can learn more by reading the Zope Developers' Guide

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.

A facility to create Perl-based Zope scripts is available for download from ActiveState . Zope does not support Perl-based scripts in the default Zope installation. 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:

```

```

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:

```

```

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* .

Advanced Acquisition

In the chapter entitled Acquisition , we introduced acquisition by containment, which we have been using throughout this chapter. In acquisition by containment, Zope looks for an object by going back up the containment heirarchy until it finds an object with the right id. In Chapter 7 we also mentioned *context acquisition* , and warned that it is a tricky subject capable of causing your brain to explode. If you are ready for exploding brains, read on.

Recall our Zoo example introduced earlier in this chapter.

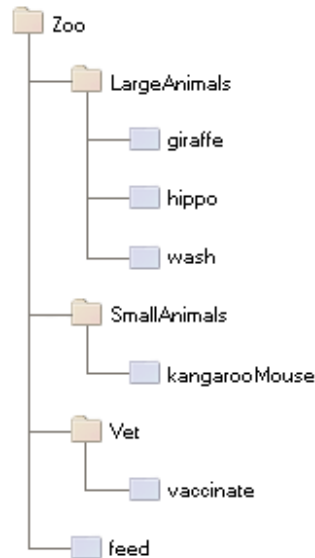


Figure 14-7 Zope Zoo Example hierarchy

We have seen how Zope uses URL traversal and acquisition to find objects in higher containers. 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. Zope allows you to combine two or more URLs into one in order to provide more acquisition context! By using acquisition, Zope will find the 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 does not 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 cannot 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.

Note that we could also have organized the URL a bit differently. *Zoo/LargeAnimals/Vet/hippo/vaccinate* would also work. The difference is the order in which the context elements are searched. In this example, we only need to get *vaccinate* from *Vet* , so all that matters is that *Vet* appears in the URL after *Zoo* and before *hippo* .

When Zope looks for a sub-object during URL traversal, it first looks for the sub-object in the current object. If it cannot find it in the current object it looks in the current object's containers. If it still cannot 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 cannot be found. If several context folders are used in the URL, they will be searched in order from *left to right* .

Context acquisition can be a very useful mechanism, and it allows you to be quite expressive when you compose URLs. The path you tell Zope to take on its way to an object will determine how it uses acquisition to look up the

object's scripts.

Note that not all scripts will behave differently depending on the traversed URL. For example, you might want your script to acquire names only from its parent containers and not from the URL context. To do so, simply use the *container* variable instead of the *context* variable in the script, as described above in the section "Using Python-based Scripts."

Context Acquisition Gotchas

Containment before context

It is important to realize that context acquisition *supplements* container acquisition. It does not *override* container acquisition.

One at a time

Another point that often confuses new users is that each element of a path "sticks" for the duration of the traversal, once it is found. Think of it this way: objects are looked up one at a time, and once an object is found, it will not be looked up again. For example, imagine this folder structure:

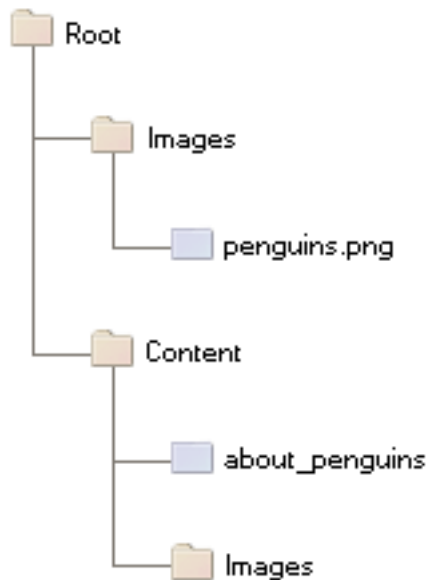


Figure 14-8 Acquisition example folder structure

Now suppose that the *about_penguins* page contains a link to *Images/penguins.png*. Shouldn't this work? Won't */Images/penguins.png* succeed when */Content/Images/penguins.png* fails? The answer is no. We always traverse from left to right, one item at a time. First we find *Content*, then *Images* within it; *penguins.png* appears in neither of those, and we have searched all parent containers of every element in the URL, so there is nothing more to search in this URL. Zope stops there and raises an error. Zope never looks in */Images* because it has already found */Content/Images*.

Readability

Context acquisition can make code more difficult to understand. A person reading your script can no longer simply look backwards up one containment heirarchy to see where an acquired object might be; many more places might be searched, all over the zope tree folder. And the order in which objects are searched, though it is consistent, can be confusing.

Fragility

Over-use of context acquisition can also lead to fragility. In object-oriented terms, context acquisition can lead to a site with low cohesion and tight coupling. This is generally regarded as a bad thing. More specifically, there are many simple actions by which an unwitting developer could break scripts that rely on context acquisition. These are more likely to occur than with container acquisition, because potentially every part of your site affects every other part, even in parallel folder branches.

For example, if you write a script that calls another script by a long and torturous path, you are assuming that the folder tree is not going to change. A maintenance decision to reorganize the folder heirarchy could require an audit of scripts in every part of the site to determine whether the reorganization will break anything.

Recall our Zoo example. There are several ways in which a zope maintainer could break the feed() script:

Inserting another object with the name of the method — This is a normal technique for customizing behavior in Zope, but context acquisition makes it more likely to happen by accident. Suppose that giraffe vaccination is controlled by a regularly scheduled script that calls *Zoo/Vet/LargeAnimals/giraffe/feed*. Suppose a content administrator doesn't know about this script and adds a DTML page called *vaccinate* in the giraffe folder, containing information about vaccinating giraffes. This new *vaccinate* object will be acquired before *Zoo/Vet/vaccinate*. Hopefully you will notice the problem before your giraffes get sick.

Calling an inappropriate path — if you visit *Zoo/LargeAnimals/hippo/buildings/visitor_reception/feed*, will the reception area be filled with hippo food? One would hope not. This might even be possible for someone who has no permissions on the reception object. Such URLs are actually not difficult to construct. For example, using relative URLs in *standard_html_header* can lead to some quite long combinations of paths.

Thanks to Toby Dickenson for pointing out these fragility issues on the zope-dev mailing list.

Calling DTML from Scripts

Often, you would want to call a *DTML Method* or *DTML Document* from a Script. For instance, a common pattern is to call a Script from an HTML form. The Script would process user input, and return an output page with feedback messages - telling the user her request executed correctly or signalling an error as appropriate.

Scripts are good at logic and general computational tasks, but ill suited for generating HTML. Therefore it makes sense to delegate the user feedback output to a DTML Method and call it from the Script.

Assume we have got an DTML Method *a_dtml_method*. We would call it from Script with:

```
# grab the method and the REQUEST from the context
dtml_method = context.a_dtml_method
REQUEST = context.REQUEST

# call the dtml method, for parameters see below
s = dtml_method(client=context, REQUEST=REQUEST, foo='bar')

# s now holds the rendered html
return s
```

Note that DTML Methods and Documents take optional *client* and *REQUEST* parameters. If a client is passed to a DTML Method, the method tries to resolve names by looking them up as attributes of the client object. By passing our context as a client, the method will look up names in that context. Also, we can pass it a REQUEST object and additional keyword arguments. The DTML Method will first try to look up variables in the keyword arguments, then the namespace, and finally in the REQUEST object. See the chapter "Advanced DTML", subchapter "DTML Namespaces," for details on namespaces, and Appendix B, API Reference for further information on DTML Methods / Documents.

Calling ZPT from Scripts

For the same reasons as outlined in the section "Calling DTML from Scripts" above, one might want to call *Page Templates* from Scripts. Assume we have this Page Template:

```
Hello <span tal:replace="options/name | default">
World
</span>
```

Calling it from a script could be done with the following Script fragment:

```
pt = context.hello_world_pt
s = pt(name="John Doe")
return s
```

The `name` parameter to the Page Template ends up in the `options/name` path expression. Of course, you can pass more than simple values to Page Templates this way. For instance, suppose we wanted to construct a list of objects and pass that to a Page Template for display. The list of objects could be constructed in an External Method. Place a file `my_extensions.py` in the `Extensions` directory, and for example add:

```
class Giraffe:
    __allow_access_to_unprotected_subobjects__ = 1

    def __init__(self, name, neck_length=None):
        self.name = name
        self.n_length=neck_length

    def neck_length(self):
        n = self.n_length
        if not n: return "unspecified"
        if type(n) == type(0.0):
            return "%s meters" % n
        return n

def giraffes(self):
    # make a list of giraffes
    glist = []
    for name, neck in (('Guido', 1.2), ('Jim', 'long'), ('Barry', None)):
        g = Giraffe(name, neck_length=neck)
        glist.append(g)
    # display the lot of them
    pt = self.display_giraffes
    return pt(giraffes=glist)
```

Add an External Method `giraffes`, module `my_extensions`, function `giraffes`. Also, add a Page Template `display_giraffes` containing the following snippet:

```
<table border="1">
  <tr>
    <th>Name</th>
    <th>Neck length</th>
  </tr>
  <tr tal:repeat="giraffe options/giraffes">
    <td tal:content="giraffe/name">name</td>
    <td tal:content="giraffe/neck_length">neck_length</td>
  </tr>
</table>
```


If you go to `Test` tab of the `giraffes` External Method, you should see a table similar to the one in the figure below.

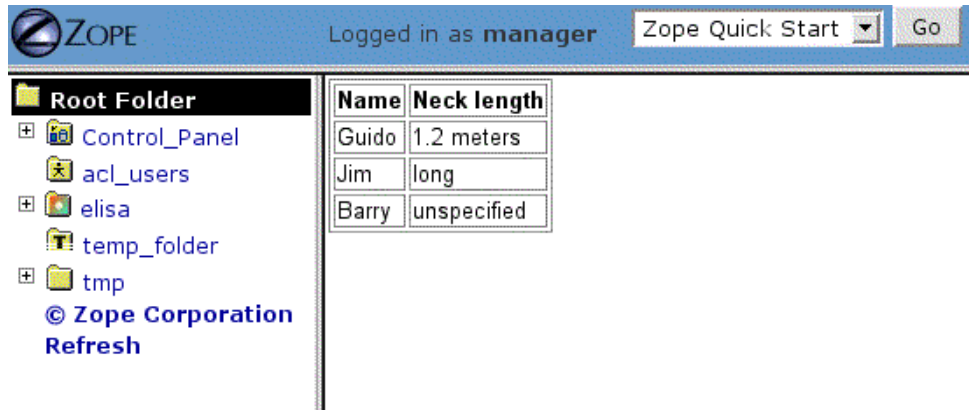


Figure 14-9 Giraffe table

In the `my_extensions` module (file `my_extensions.py`) we define a class `Giraffes`. The `__allow_access_to_unprotected_subobjects__ = 1` statement tells Zope that it is okay to grant access on giraffes to everyone - use only when you actually want this!

Then, in the `giraffes` function, we create some giraffe objects using hardcoded values, and hand the list of giraffes to a Page Template, which iterates through the list to display their data in a table.

Again, we use the `options` Page Template variable to access the list members, and access data and method attributes of the giraffe objects using path expressions like `giraffe/name`. The `neck_length` method gets called automatically in this process. You would want to use python expression syntax instead if you need to pass parameters to the giraffe objects methods, eg. use something like `python:giraffe.neck_length()`. See the chapters on "Zope Page Templates" and "Advanced Page Templates" for more details on path and python expressions.

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="form_action">
Name of Hippo <input type="text" name="name"><br>
Age of Hippo <input type="text" name="age"><br>
<input type="submit">
</form>
```

You can easily process this form with a script named `form_action` that includes `name` and `age` in its parameter list:

```
## Script (Python) "form_action"
##parameters=name, age
##
"Process form"
age=int(age)
message= 'This hippo is called %s and is %d years old' % (name, age)
if age < 18:
    message += '\n %s is not old enough to drive!' % name
return message
```

There is 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 checkboxes 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 .

In the Python script given above, there is a subtle problem. You are probably expecting an integer rather than a string for age, but all form variables are passed as strings. Perl takes care of such things automagically, but Python does not. You could manually convert the string to an integer using the Python `int` built-in:

```
age=int(age)
```

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 itself:

```
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. This process is called *marshalling* . If the user of your form types something that cannot be converted to an integer (such as "22 going on 23") then Zope will raise an exception as shown in the figure below.

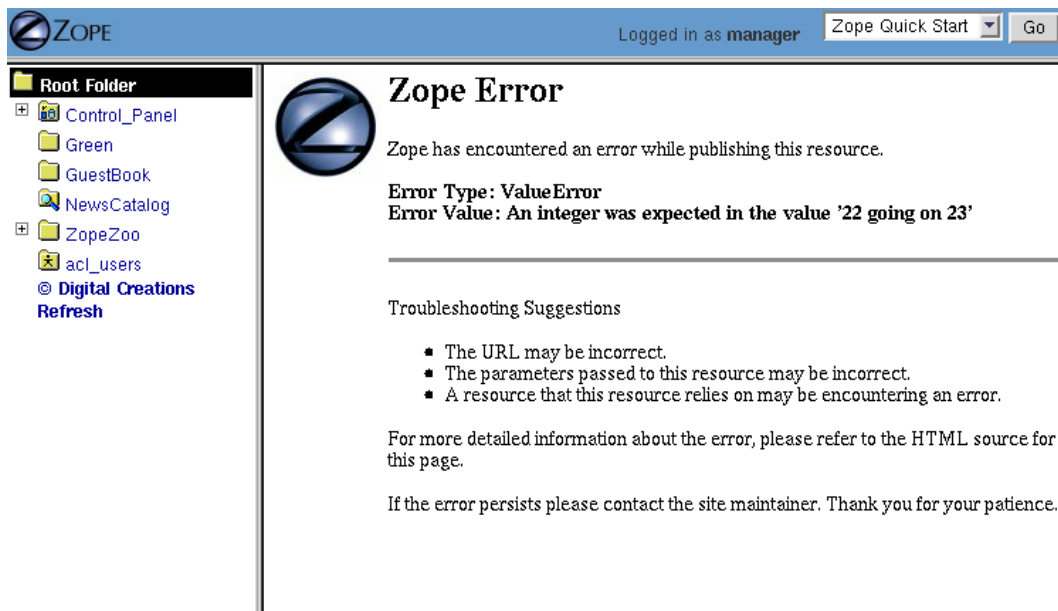


Figure 14-10 Parameter conversion error

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 form variable, which is a string, into another specific type. You may recognize these converters from the chapter entitled Using Basic Zope Objects , in which 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 the attributes *fname*, *lname* and *age*. 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 is 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. Note the difference between the *records* converter and the *list:record* converter: the former would create a list of records, whereas the latter would produce a single record whose attributes *fname* and *lname* would each be a list of values.

The order of combined modifiers does not matter; for example, *int:list* is identical to *list:int*.

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 — Appends the attribute value to the original form action of the form. This is mostly useful for the case in which 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. A synonym for *action* is *method*.

default_action — Appends the attribute value to the original action of the form when no other *action* converter is used.

Here's an example form that uses action converters:

```
<form action="employeeHandlers">
<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 in the *employeeHandlers* folder, 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.

Returning Values from Scripts

Scripts have their own variable scope. In this respect, scripts in Zope behave just like functions, procedures, or methods in most programming languages. If you call a script *updateInfo*, for example, and *updateInfo* assigns a value to a variable *status*, then *status* is local to your script -- it gets cleared once the script returns. To get at the value of a script variable, we must pass it back to the caller with a *return* statement.

Here is an example of how one might call a script from DTML and use a value returned from the script:

```
<dtml-let status="updateInfo(color='brown', pattern='spotted')">
  <dtml-if expr="status == 0">
    Data updated successfully
  <dtml-else>
    An error occurred! The error status was: <dtml-var status>
  </dtml-if>
</dtml-let>
```

Scripts can only return a single object. If you need to return more than one value, put them in a dictionary and pass that back.

Suppose you have a Python script *compute_diets* out of which you want to get values:

```
## Script (Python) "compute_diets"
d = {'fat': 10,
     'protein': 20,
     'carbohydrate': 40,
     }
return d
```

The values would, of course, be calculated in a real application; in this simple example we will just hardcode some numbers.

You could call this script from DTML like this:

```
<dtml-with compute_diets mapping>
  This animal needs
  <dtml-var fat>kg fat,
  <dtml-var protein>kg protein, and
  <dtml-var carbohydrate>kg carbohydrates.
</dtml-with>
```

Note the *mapping* attribute to the dtml-with tag - it tells DTML to expect a mapping (a dictionary in our case) instead of an object.

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*.

The chapter entitled 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.

Security Restrictions of Script (Python)

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 the chapter entitled Users and Security for more information). In addition, you cannot access objects whose names begin with an underscore, since Zope considers these objects to be private. Finally, you can define classes in scripts but it is not really practical to do so, because you are not allowed to access attributes of these classes! Even if you were allowed to do so, the restriction against using objects whose names begin with an underscore would prevent you from using your class's `__init__` method. If you need to define classes, use *External Methods* (see below) or *Zope Products* (see the Zope Developers Guide for more information about creating Products). You may, however, define functions in scripts, although it is rarely useful or necessary to do so. In practice, a Script in Zope is treated as if it were a single method of the object you wish to call it on.

Writing limits — In general you cannot directly change Zope object attributes using scripts. You should call the appropriate methods from the Zope API instead.

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 should not grant access to scripts to untrusted people.

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 the figure below.

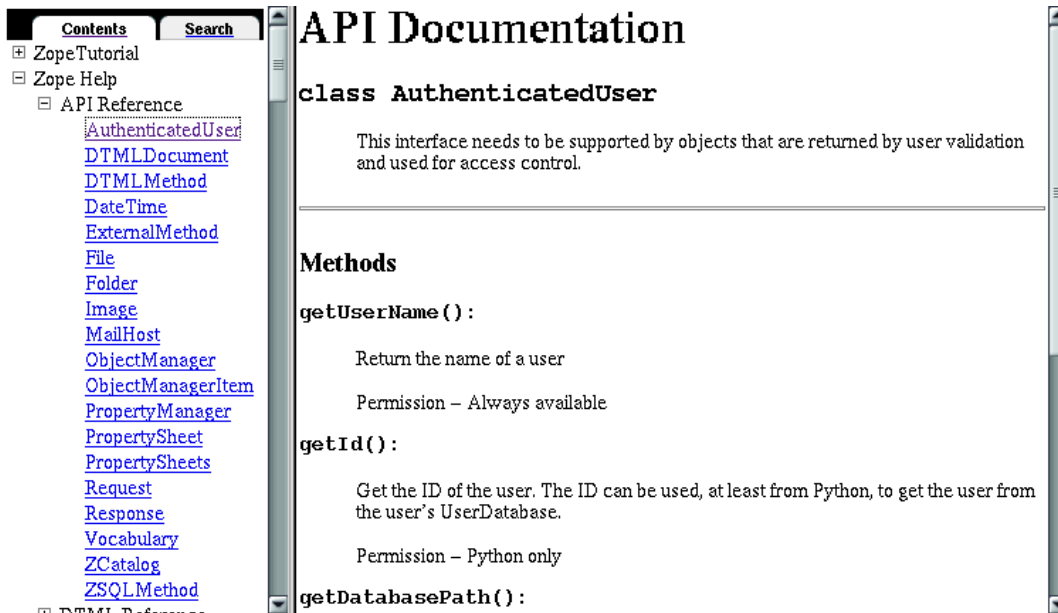


Figure 14-11 Zope API Documentation

Suppose you would 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.

To get you started, and whet your appetite, we will go through some example Python scripts that demonstrate how you can use the Zope API.

Get all objects in a folder

The `objectValues()` method returns a list of objects contained in a folder. If the context happens not to be a folder, nothing is returned:

```
objs = context.objectValues()
```

Get the id of an object

The id is the "handle" to access an object, and is set at object creation:

```
id = context.getId()
```

Note that there is no `setId()` method - you have to either use the ZMI to rename them, set their `id` attribute via security-unrestricted code, or use the `manage_renameObject` or `manage_renameObjects` API methods exposed upon the container of the object you want to rename.

Get the Zope root folder

The root folder is the top level element in the Zope object database:

```
root = context.getPhysicalRoot()
```

Get the physical path to an object

The `getPhysicalPath()` method returns a list contained the ids of the object's containment heirarchy :

```
path_list = context.getPhysicalPath()
path_string = "/".join(path_list)
```

Get an object by path

`restrictedTraverse()` is the complement to `getPhysicalPath()`. The path can be absolute - starting at the Zope root - or relative to the context:

```
path = "/Zoo/LargeAnimals/hippo"
hippo_obj = context.restrictedTraverse(path)
```

Change the content of an DTML Method or Document

You can actually change the content (and title) of a DTML Method or Document, exactly as if you edited it in the Zope management interface, by using its `manage_edit()` method:

```
# context has to be a DTML method or document!
context.manage_edit('new content', 'new title')
```

Change properties of an object

You can use the `manage_changeProperties` method of any Zope object to change its properties:

```
# context may be any kind of Zope object
context.manage_changeProperties({'title':'Another title'})
```

Get a property

`getProperty()` returns a property of an object. Many objects support properties (those that are derived from the `PropertyManager` class), the most notable exception being DTML Methods, which do not:

```
pattern = context.getProperty('pattern')
return pattern
```


Change properties of an object

The object has to support properties and the property must exist:

```
values = {'pattern' : 'spotted'}
context.manage_changeProperties(values)
```

Execute a DTML Method or DTML Document

This executes a DTML Method or Document and returns the result. Note that DTML Methods and Documents take optional *client* and *REQUEST* parameters. If a client is passed to a DTML Method, the method tries to resolve names by looking them up as attributes of the client object. By passing our context as a client, the method will look up names in that context. Also, we can pass it a REQUEST object and additional keyword arguments - the DTML Method will first try to resolve names in them. See the chapter entitled Advanced DTML , section "DTML Namespaces," for details:

```
dtml_method = context.a_dtml_method
s = dtml_method(client=context, REQUEST={}, foo='bar')
return s
```

Traverse to an object and add a new property

We get an object by its absolute path and add a property `weight` , and set it to some value. Again, the object must support properties for this to work. We introduce another neat trick in this example. Long method names can make lines in your scripts long and hard to read, so you can assign a shorter name for the method before using it:

```
path = "/Zoo/LargeAnimals/hippo"
hippo_obj = context.restrictedTraverse(path)
add_method = hippo_obj.manage_addProperty
add_method('weight', 500, 'int')
```

Add a new object to the context

Scripts can add objects to folders, just like you do in the Zope management interface. The context has to be a folderish object (i.e. a folder or another object derived from `ObjectManager`). The general pattern is:

```
context.manage_addProduct['PackageName'].manage_addProductName(id)
```

`manage_addProduct` is a mapping in which we can look up a *dispatcher* - an object which gives us the necessary *factory* for creating a new object in the context. For most of the built-in Zope classes, the `PackageName` is `OFSP` , and the factory method is named after the product class itself. Once you have the factory method, you must pass it whatever arguments are needed for adding an object of this type. Some examples will make this clear.

Let's add a DTML Method to a folder:

```
add_method = context.manage_addProduct['OFSP'].manage_addDTMLMethod
add_method('object_id', file="Initial contents of the DTML Method")
```

For any other product class that comes with Zope, we need only change the factory method and its arguments.

DTML Methods — `manage_addDTMLMethod`

DTML Documents — `manage_addDTMLDocument`

Images — `manage_addImage`

Files — `manage_addFile`

Folders — *manage_addFolder*

UserFolders — *manage_addUserFolder*

Version — *manage_addVersion*

To get a dispatcher for add-on Products which you download and install, replace `OFSP` with the directory which contains the product code. For example, if you have installed the famous Boring product, you could add one like so:

```
add_method = context.manage_addProduct['Boring'].manage_addBoring
add_method(id='test_boring')
```

If the product author has been conscientious, the process for adding new instances of their product will be documented; but it will always look something like the above examples.

DTML versus Python versus Perl versus Page Templates

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.

For presentation, Perl and Python should *not* be used; the choice then becomes whether to use DTML or ZPT.

Just for the sake of comparison, here is a simple presentational script suggested by Gisle Aas, the author of Perl-based Scripts, in four different languages.

In DTML:

```
<dtml-in objectValues>
  <dtml-var getId>: <dtml-var sequence-item>
</dtml-in>
done
```

In ZPT:

```
<div tal:repeat="item here/objectValues"
      tal:replace="python:'%s: %s\n' % (item.getId(), str(item))" />
```

In Python:

```
for item in context.objectValues():
    print "%s: %s" % (item.getId(), item)
print "done"
return printed
```

In Perl:

```
my $context = shift;
my @result;

for ($context->objectValues()) {
    push(@result, join(":", $_->getId(), $_));
}
join("\n", @result, "done");
```

Despite the fact that Zope is implemented in Python, it sometimes (for better or worse) 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 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 employee 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) {
    ioex.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-passwd=SecretPhrase http://www.zopezope.org/Lions/feed
```

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 -nv --spider --http_user=ZooKeeper --http_pass=SecretPhrase http://www.zopezoo.org/Lions/feed
```

The only difference between using *cron* and calling *wget* manually is that you should use the *nv* 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 .

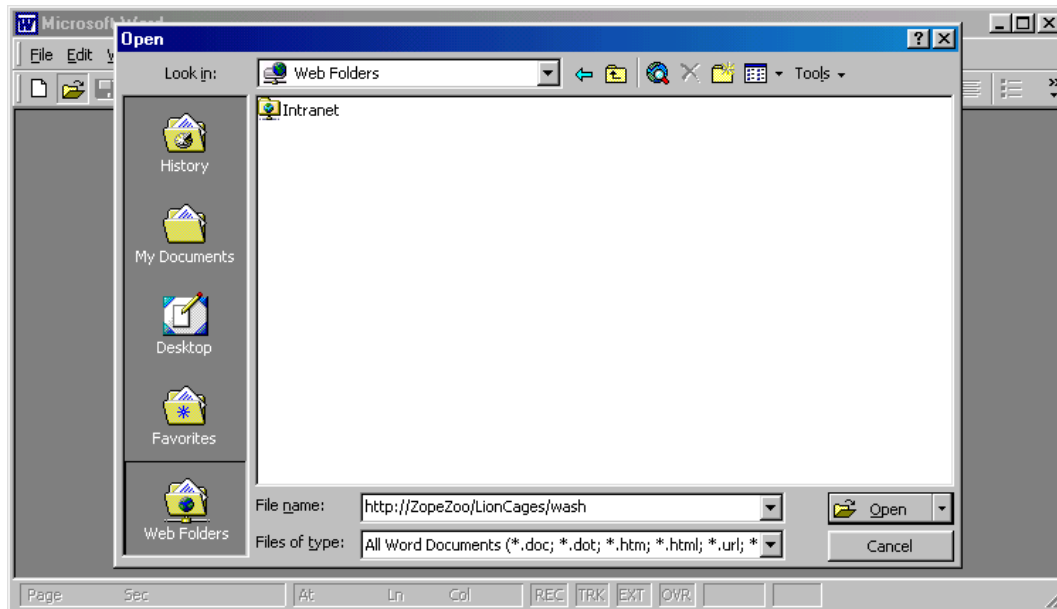


Figure 14-12 Calling a URL with Microsoft Word

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 easily script Zope with almost any network-aware system.

Conclusion

Zope provides scripting with Python, and optionally with Perl. With scripts you can control Zope objects and glue together your application's logic, data, and presentation. You can programmatically manage objects in your Zope folder hierarchy by using the Zope API. You can also perform serious programming tasks such as image processing and XML parsing.

Zope Services

Some Zope objects are *service* objects. *Service* objects provide various kinds of support to your "domain-specific" content, logic, and presentation objects. They help solve fundamental problems that many others have experienced when writing applications in Zope.

Access Rule Services

Access Rules make it possible to cause an action to happen any time a user "traverses" a Folder in your Zope site. When a user's browser submits a request for a URL to Zope which has a Folder's name in it, the Folder is "looked up" by Zope during object publishing. That action (the lookup) is called *traversal*. Access Rules are arbitrary bits of code which effect the environment in some way during Folder traversal. They are easiest to explain by way of an example.

In your Zope site, create a Folder named "accessrule_test". Inside the accessrule_test folder, create a Script (Python) object named `access_rule` with two parameters: `container` and `request`. Give the `access_rule` Script (Python) the following body:

```
useragent = request.get('HTTP_USER_AGENT', '')
if useragent.find('Windows') != -1:
    request.set('OS', 'Windows')
elif useragent.find('Linux') != -1:
    request.set('OS', 'Linux')
else:
    request.set('OS', 'Non-Windows, Non-Linux')
```

This Script causes the traversal of the `accessrule_test` folder to cause a new variable named `OS` to be entered into the `REQUEST`, which has a value of `Windows`, `Linux`, or `Non-Windows, Non-Linux` depending on the user's browser.

Save the `access_rule` script and revisit the `accessrule_test` folder's *Contents* view. Choose *Set Access Rule* from the add list. In the *Rule Id* form field, type `access_rule`. Then click *Set Rule*. A confirmation screen appears claiming that "`access_rule` is now the Access Rule for this object". Click "OK". Notice that the icon for the `access_rule` Script (Python) has changed, denoting that it is now the access rule for this Folder.

Create a DTML Method named `test` in the `accessrule_test` folder with the following body:

```
<dtml-var standard_html_header>
<dtml-var REQUEST>
<dtml-var standard_html_footer>
```

Save the `test` DTML Method and click its "View" tab. You will see a representation of all the variables that exist in the `REQUEST`. Note that in the **other** category, there is now a variable named "OS" with (depending on your browser platform) either `Windows`, `Linux` or `Non-Linux, Non-Windows`).

Revisit the `accessrule_test` folder and again select *Set Access Rule* from the add list. Click the *No Access Rule* button. A confirmation screen will be displayed stating that the object now has no Access Rule.

Visit the `test` script you created previously and click its *View* tab. You will notice that there is now no "OS" variable listed in the request because we've turned off the Access Rule capability for `access_rule`.

Access Rules have many potential creative uses. For example, a ZopeLabs recipe submitted by xzc shows how to restrict a specific user agent from accessing a particular Zope object. Another tip from runyaga shows how to use an access rule to restrict management access in a CMF site for non-manager users. Another recipe by ivo tells us how to transparently delegate requests for an object to another object using an Access Rule.

Access Rules don't need to be Script (Python) objects, they may also be DTML Methods or External Methods.

Temporary Storage Services

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

1. Everything contained in a Temporary Folder disappears when you restart Zope. (A Temporary Folder's contents are stored in RAM).
2. You cannot undo actions taken to objects stored 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.

Version Services

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.

NOTE: Using versions via the Zope Management Interface requires that your browser supports and accepts cookies from the Zope server.

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 the figure below.

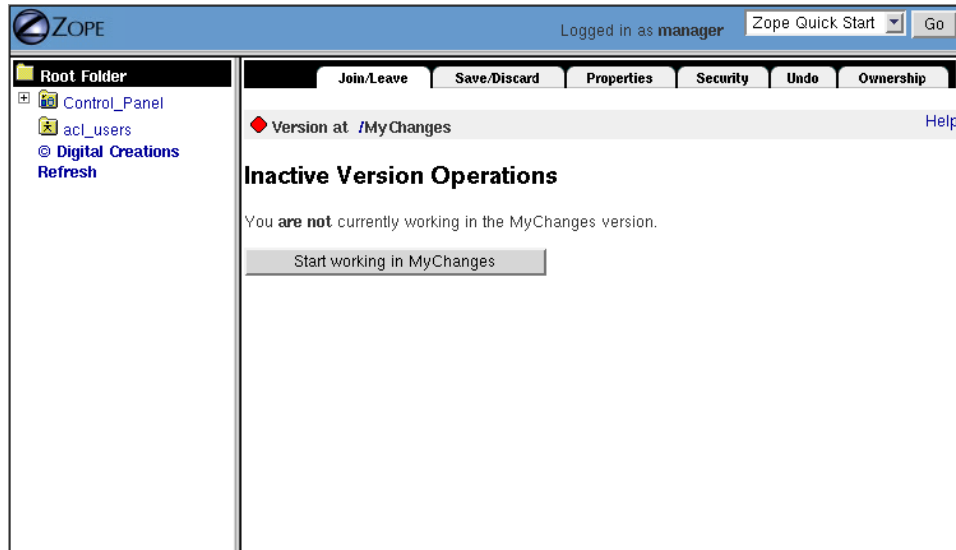


Figure 6-1 Joining a Version

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 the figure below.

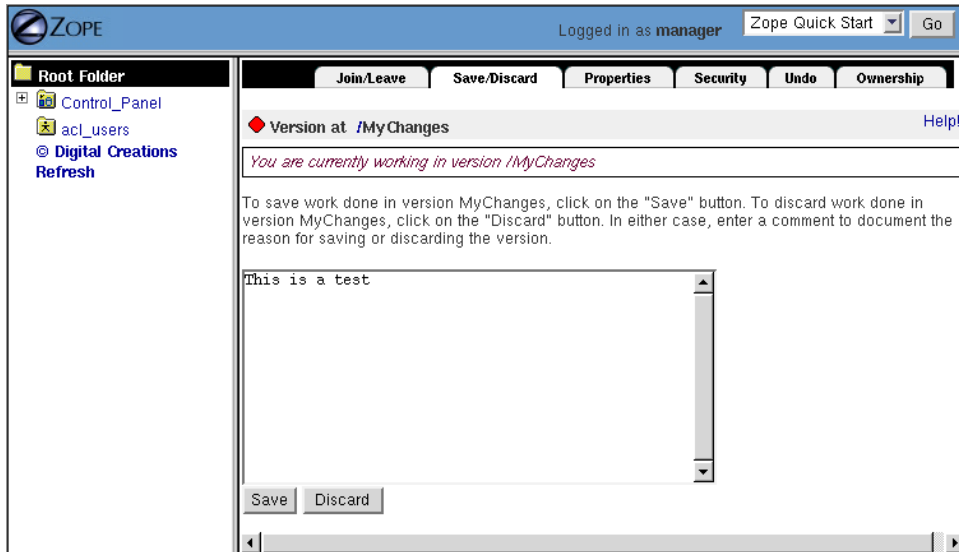


Figure 6-2 Committing Version changes.

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.

Caveat: Versions and ZCatalog

ZCatalog is Zope's indexing and searching engine, covered in depth in the chapter entitled Searching and Categorizing Content .

Unfortunately, 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.

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 propagate 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 itself. This makes the catalog and versions get along poorly. As a rule, versions should not be used in applications that use the catalog.

Caching Services

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 Script (Python), 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 an Object

Caching any sort of cacheable object is fairly straightforward. First, before you can cache an object 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 yesterday's forecast and the current temperature, respectively (for more information on SQL Methods, see the chapter entitled 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.

Outbound Mail Services

Zope comes with an object that is used to send outbound e-mail, usually in conjunction with the DTML `sendmail` tag, described more in the chapter entitled 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 the chapter entitled Variables and Advanced DTML , but we provide a simple example below. In your root folder, create a DTML Method named `send_mail` with a body that looks like the following:

```
<dtml-sendmail>
From: me@nowhere.com
To: you@nowhere.com
Subject: Stop the madness!

Take a day off, you need it.

</dtml-sendmail>
```

Ensure that all the lines are flush against the left side of the textarea for proper function. When you invoke this DTML Method (perhaps by visiting its *View* tab), it will use your newly-created MailHost to send an admonishing mail to "you@nowhere.com". Substitute your own email address to try it out.

The API for MailHost objects also allows you to send mail from Script (Python) objects and External Methods. See the Zope MailHost API in the Zope help system at Zope Help -> API Reference -> MailHost for more information about the interface it provides.

Error Logging Services

The *Site Error Log* object, typically accessible in the Zope root under the name `error_log` , provides debugging and error logging information in real-time. When your site encounters an error, it will be logged in the Site Error Log, allowing you to review (and hopefully fix!) the error.

Options settable on a Site Error Log instance include:

Number of exceptions to keep — keep 20 exceptions by default, rotating "old" exceptions out when more than 20 are stored. Set this to a higher or lower number as you like.

Copy exceptions to the event log — If this option is selected, the site error log object will copy the text of exceptions that it receives to the "event log" facility, which is typically controlled by the `EVENT_LOG_FILE` environment variable. For more information about this environment variable, see the chapter entitled *Installing and Starting Zope* .

Virtual Hosting Services

For detailed information about using virtual hosting services in Zope, see the chapter entitled *Virtual Hosting Services* .

Searching and Indexing Services

For detailed information about using searching and indexing services in Zope to index and search a collection of documents, see the chapter entitled *Searching and Categorizing Content* .

Sessioning Services

For detailed information about using Zope's "sessioning" services to "keep state" between HTTP requests for anonymous users, see the chapter entitled *Sessions* .

Internationalization Services

This section of the document needs to be expanded. For now, please see documentation for Zope 2.6+ wrt Unicode and object publishing at <http://www.zope.org/Members/htrd/howto/unicode-zdg-changes> and <http://www.zope.org/Members/htrd/howto/unicode> .

Searching and Categorizing Content

The ZCatalog 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 ZCatalog to organize collections of objects.

The ZCatalog supports a rich query interface. You can perform full text searching, and can search multiple indexes at once. In addition, the ZCatalog 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.

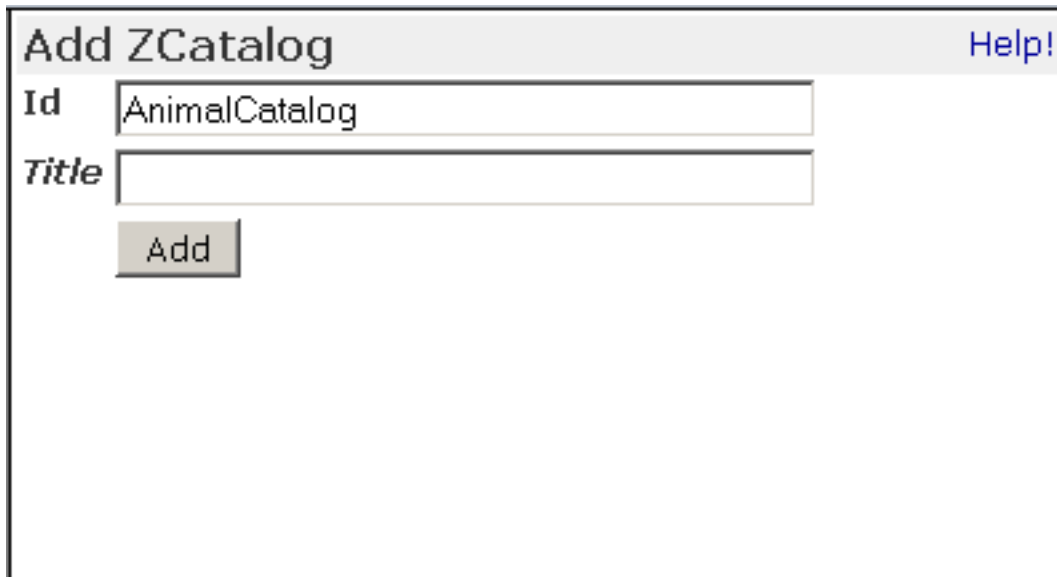
Getting started with Mass Cataloging

Let's take a look at how to use the ZCatalog to search documents. Cataloging a bunch of objects all at once is called *mass Cataloging*. Mass Cataloging involves three steps:

- Creating a ZCatalog
- Creating Indexes
- Finding objects and cataloging them
- Creating a web interface to search the ZCatalog.

Creating a ZCatalog

Choose *ZCatalog* from the product add list to create a ZCatalog object within a subfolder named `zoo`. This takes you to the ZCatalog add form, as shown in the figure below.



The screenshot shows a web form titled "Add ZCatalog". In the top right corner of the form area, there is a blue link labeled "Help!". The form contains two text input fields. The first field is labeled "Id" and contains the text "AnimalCatalog". The second field is labeled "Title" and is currently empty. Below these two fields is a rectangular button labeled "Add".

Figure 16-1 ZCatalog add form

The Add form asks you for an *Id* and a *Title*. Give your ZCatalog the Id `AnimalCatalog` and click *Add* to create your new ZCatalog. The ZCatalog icon looks like a folder with a small magnifying glass on it. Select the *AnimalCatalog* icon to see the *Contents* view of the ZCatalog.

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. Although a ZCatalog is like a normal Zope folder, this does not imply that the objects contained within it are automatically searchable. A ZCatalog can catalog objects at any level of your site, and it needs to be told exactly which ones to index.

Creating Indexes

In order to tell Zope what to catalog and where to store the information, we need to create a *Lexicon* and an *Index*. A *Lexicon* is necessary to provide word storage services for full-text searching, and an *Index* is the object which stores the data necessary to perform fast searching.

In the contents view of the *AnimalCatalog* ZCatalog, choose *ZCTextIndex Lexicon*, and give it an id of `zooLexicon`

Figure 16-2 ZCTextIndex Lexicon add form

Now we can create an index that will record the information we want to have in the ZCatalog. Click on the *Indexes* tab of the ZCatalog. A drop down menu lists the available indexes. Choose *ZCTextIndex*; in the add form fill in the id *zooTextIdx*. Fill in *PrincipiaSearchSource* in the "Field name" input. This tells the ZCTextIndex to index the body text of the DTML Documents (*PrincipiaSearchSource* is an API method of all DTML Document and Method objects). Note that *zooLexicon* is preselected in the *Lexicon* menu.

Figure 16-3 ZCTextIndex add form

To keep this example short we will skip over some of the options presented here. In the section on indexes below, we will discuss this more thoroughly.

Additionally, we will have to tell the ZCatalog which attributes of each cataloged object that it should store directly. These attributes are called *Metadata*. For now, just go to the *Metadata* tab of the ZCatalog and add *id* and *title*.

Finding and Cataloging Objects

Now that you have created a ZCatalog and an Index, 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 along-side the *AnimalCatalog* object (within the same folder that contains the *AnimalCatalog* ZCatalog) that contain information about reptiles and amphibians.

The first should have an *id* of "chilean_frog", a title "Chilean four-eyed frog" and its body text should read something like this:

```
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.
```

For the second, fill in an *id* of "carpet_python" and a title of "Carpet Python"; its body text could be:

```
*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 *AnimalCatalog* 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 *AnimalCatalog* 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. For example, if your ZCatalog is located at `/Zoo/AnimalCatalog`, then the `/Zoo` folder and all its subfolders will get searched.

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 ZCatalog 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 fourth 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 *AnimalCatalog* ZCatalog and select *Z Search Interface* from the add list. Select the *AnimalCatalog* ZCatalog as the searchable object, as shown in the figure below.

The screenshot shows a web-based dialog box titled "Add Search Interface". It contains the following elements:

- Title Bar:** "Add Search Interface" with a "Help!" link on the right.
- Text:** "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." and "In the form below, *searchable 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."
- Form Fields:**
 - "Select one or more searchable objects": A dropdown menu with "AnimalCatalog" selected.
 - "Report Id": A text input field containing "SearchReport".
 - "Report Title": An empty text input field.
 - "Report Style": A dropdown menu with "Tabular" selected.
 - "Search Input Id": A text input field containing "SearchForm".
 - "Search Input Title": An empty text input field.
- Options:** Two radio buttons: "Generate DTML Methods" (unselected) and "Generate Page Templates" (selected).
- Buttons:** An "Add" button at the bottom.

Figure 16-4 Creating a search form for a ZCatalog

Name the *Report Id* "SearchResults", the *Search Input Id* "SearchForm", select "Generate Page Templates" and click *Add*. This will create two new Page Templates in the *AnimalCatalog* ZCatalog named *SeachForm* and *SearchResults*.

These objects are *contained in* the ZCatalog, but they are not *cataloged by* the ZCatalog. The *AnimalCatalog* has only cataloged DTML Documents. The search Form and Report templates are just a user interface to search the animal documents in the ZCatalog. You can verify this by noting that the search and report forms are not listed in the *Cataloged Objects* tab.

To search the *AnimalCatalog* ZCatalog, select the *SearchForm* template and click on its *Test* tab.

By typing words into the *ZooTextIdx* form element you can search all of the documents cataloged by the *AnimalCatalog* ZCatalog. For example, type in the word "Reptiles". The *AnimalCatalog* 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 "reptiles OR amphibians". Search results for this query should include both the Chilean four-eyed Frog and the carpet python. Congratulations, you have successfully created a ZCatalog, cataloged content into it and searched it through the web.

Configuring ZCatalogs

The ZCatalog is capable of much more powerful and complex searches than the one you just performed. Let's take a look at how the ZCatalog stores information. This will help you tailor your ZCatalogs 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 2.6, indexes can be added and removed from a ZCatalog using a new, "pluggable" index interface as shown in the figure below:

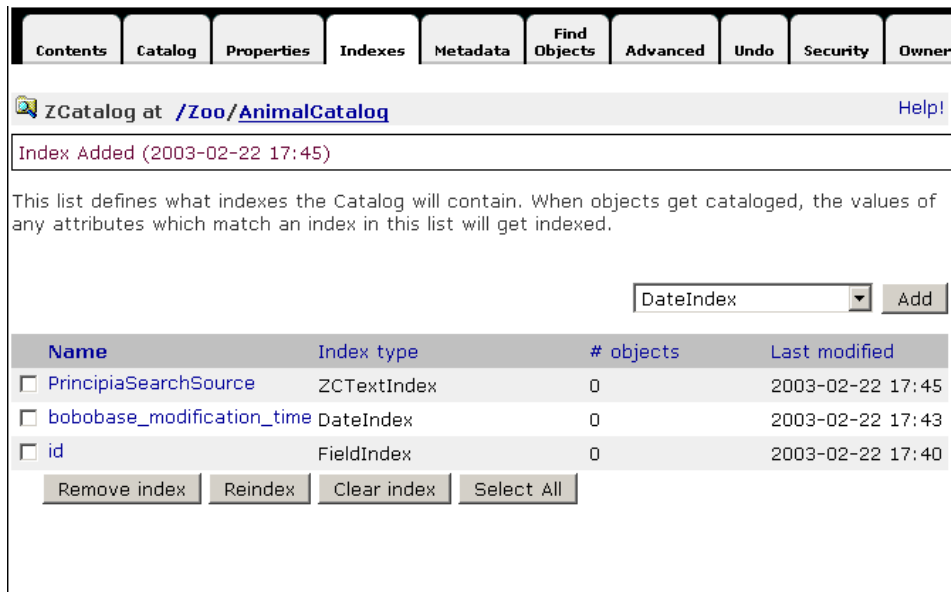


Figure 16-5 Managing indexes

Each index has a name, like *PrincipiaSearchSource*, and a type, like *ZCTextIndex*.

When you catalog an object the ZCatalog uses each index to examine the object. The ZCatalog 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 ZCatalog calls each document's *PrincipiaSearchSource* method and records the results in its *PrincipiaSearchSource* index. If the ZCatalog 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 eight kinds of indexes:

ZCTextIndex — 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 `/Zoo/Animals` .

TopicIndex — Searches among *FilteredSets*; each set contains the document IDs of documents which match the set's filter expression. Use this kind of index to optimize frequently-accessed searches.

DateIndex — A subclass of *FieldIndex*, optimized for date-time values. Use this index for any field known to be a date or a date-time.

DateRangeIndex — Searches objects based on a pair of dates / date-times. Use this index to search for objects which are "current" or "in effect" at a given time.

TextIndex — Old version of a full-text index. Only provided for backward compatibility, use *ZCTextIndex* instead.

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. For a *ZCTextIndex*, you will also need a *ZCTextIndex Lexicon* object in your *ZCatalog* - see below for details.

To remove an index from a *ZCatalog*, 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 *Metadata Table* . The *Metadata 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 *ZCatalog*'s indexes. Indexes allow you to search; meta-data allows you to report search results.

The *Metadata 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 *Metadata Table* column called *Title* , then your report forms can use this information to show the titles of your objects that are returned in search results instead of requiring that you actually obtain the object to show its title.

To add a new *Metadata Table* column, type in the name of the column on the *Metadata Table* view and click *Add* . To remove a column from the *Metadata 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 *ZCatalog*.

Searching ZCatalogs

You can search a *ZCatalog* by passing it search terms. These search terms describe what you are looking for in one or more indexes. The *ZCatalog* 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 *ZCatalog* 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 ZCatalog (the Form/Action pattern is discussed in the chapter entitled Advanced Page Templates). 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 ZCatalogs are queried and how you can customize and extend your search interface.

Suppose you have a ZCatalog that holds news items named `NewsCatalog` . Each news item has `content` , an `author` and a `date` attribute. Your ZCatalog has three indexes that correspond to these attributes, namely "contentTextIdx", "author" and "date". The contents index is a ZCTextIndex, and the author and date indexes are a FieldIndex and a DateIndex. For the ZCTextIndex you will need a ZCTextIndexLexicon, and to display the search results in the `Report` template, you should add the `author` , `date` and `absolute_url` attributes as Metadata. Here is a search form that would allow you to query such a ZCatalog:

```
<html><body>
<form action="Report" method="get">
<h2 tal:content="template/title_or_id">Title</h2>
Enter query parameters:<br><table>
<tr><th>Author</th>
<td><input name="author" width=30 value=""></td></tr>
<tr><th>Content</th>
<td><input name="contentTextIdx" 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" name="SUBMIT" value="Submit Query">
</td></tr>
</table>
</form>
</body></html>
```

This form consists of three input boxes named `contentTextIdx` , `author` , and `date` . These names must match the names of the ZCatalog's indexes for the ZCatalog to find the search terms. Here is a report form that works with the search form:

```
<html>
<body tal:define="searchResults here/NewsCatalog;">
<table border>
<tr>
<th>Item no.</th>
<th>Author</th>
<th>Absolute url</th>
<th>Date</th>
</tr>
<div tal:repeat="item searchResults">
<tr>
<td>
<a href="link to object" tal:attributes="href item/absolute_url">
#<span tal:replace="repeat/item/number">
search item number goes here
</span>
</a>
</td>
<td><span tal:replace="item/author">author goes here</span></td>
<td><span tal:replace="item/date">date goes here</span></td>
</tr>
</div>
</table>
</body></html>
```

There are a few things going on here which merit closer examination. The heart of the whole thing is in the definition of the `searchResults` variable:

```
<body tal:define="searchResults here/NewsCatalog;">
```

This calls the `NewsCatalog` ZCatalog. Notice how the form parameters from the search form (`contentTextIdx` , `author` , `date`) are not mentioned here at all. Zope automatically makes sure that the query parameters from the search form are given to the ZCatalog. All you have to do is make sure the report form calls the ZCatalog. Zope locates the search terms in the web request and passes them to the ZCatalog.

The ZCatalog 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 ZCatalog will only return records that match both the author and the contents.

ZSQL Record objects have an attribute for every column in the database table. Record objects for ZCatalogs work very similarly, except that a ZCatalog Record object has an attribute for every column in the Metadata Table. In fact, the purpose of the Metadata Table is to define the schema for the Record objects that ZCatalog queries return.

Searching from Python

Page Templates make querying a ZCatalog from a form very simple. For the most part, Page Templates will automatically make sure your search parameters are passed properly to the ZCatalog.

Sometimes though you may not want to search a ZCatalog from a web form; some other part of your application may want to query a ZCatalog. 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 ZCatalog with the currentfolder's id:

```
## Script (Python) "relevantSectionNews"
##
""" Returns news relevant to the current folder's id """
id=context.getId()
return context.NewsCatalog({'contentTextIdx' : id})
```

This script queries the `NewsCatalog` by calling it like a method. ZCatalogs 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 `contentTextIdx` index will be queried for all news items that contain the name of the current Folder. To use this in your sidebar place you could insert this snippet where appropriate in the main ZopeZoo Page Template:

```
...
<ul>
  <li tal:repeat="item here/relevantSectionNews">
    <a href="news link" tal:attributes="href item/absolute_url">
      <span tal:replace="item/title">news title</span>
    </a>
  </li>
</ul>
...
```

This template assumes that you have defined `absolute_url` and `title` as Metadata columns in the `NewsCatalog` . 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 ZCatalog supports eight types of indexes. Let's examine these indexes more closely to understand what they are good for and how to search them.

Searching ZCTextIndexes

A ZCTextIndex is used to index text. After indexing, you can search the index for objects that contain certain words. ZCTextIndexes support a rich search grammar for doing more advanced searches than just looking for a word.

Boolean expressions

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 NOT. A synonym for NOT is a leading hyphen:

```
word1 -word2
```

which would search for occurrences of "word1" but would exclude documents which contain "word2". A sequence of words without operators implies AND. A search for "carpet python snakes" translates to "carpet AND python AND snakes".

Parentheses

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".

Wild cards

Search for wild cards like:

```
Z*
```

which returns all words that begin with "Z", or:

```
Zop?
```

which returns all words that begin with "Zop" and have one more character - just like in a Unix shell. Note though that wild cards cannot be at the beginning of a search phrase. "?ope" is an illegal search term and will be ignored.

Phrase search

Double-quoted text implies phrase search, for example:

```
"carpet python" OR frogs
```

will search for all occurrences of the phrase "carpet python" or of the word "frogs"

All of these advanced features can be mixed together. For example:

```
((bob AND uncle) 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.

Similarly, a search for:

```
snakes OR frogs -"carpet python"
```

will return all objects which contain the word "snakes" or "frogs" but do not contain the phrase "carpet python".

Querying a ZCTextIndex 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 ZCTextIndex 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-catastrophic news """
id=context.getId()
return context.NewsCatalog(
    {'contentTextIdx' : id + ' -catastrophic'}
)
```

ZCTextIndexes are very powerful. When mixed with the Automatic Cataloging pattern described later in the chapter, they give you the ability to automatically full-text search all of your objects as you create and edit them.

Lexicons

Lexicons are used by ZCTextIndexes. Lexicons process and store the words from the text and help in processing queries.

Lexicons can:

Normalize Case — Often you want search terms to be case insensitive, eg. a search for "python", "Python" and "pYTHON" should return the same results. The lexicons' *Case Normalizer* does exactly that.

Remove stop words — Stop words are words that are very common in a given language and should be removed from the index. They would only cause bloat in the index and add little information.

Split text into words — A splitter parses text into words. Different texts have different needs of word splitting - if you are going to process HTML documents, you might want to use the HTML aware splitter which effectively removes HTML tags. On the other hand, if you are going to index plain text documents *about* HTML, you don't want to remove HTML tags - people might want to look them up. Also, an eg. chinese language document has a different concept of words and you might want to use a different splitter.

The Lexicon uses a pipeline architecture. This makes it possible to mix and match pipeline components. For instance, you could implement a different splitting strategy for your language and use this pipeline element in conjunction with the standard text processing elements. Implementing a pipeline element is out of the scope of this book; for examples of implementing and registering a pipeline element see eg. `lib/python/Products/ZCTextIndex/Lexicon.py`. A pipeline element should conform to the `IPipelineElement` interface.

To create a ZCTextIndex, you first have to create a Lexicon object. Multiple ZCTextIndexes can share the same lexicon.

Searching Field Indexes

FieldIndexes differ slightly from ZCTextIndexes. A ZCTextIndex 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 a `FieldIndex` `author`. With the existing search form, this field is not very useful. Unless you know exactly the name of the author you are looking for, 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.

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 and replace the original `author` input box with something a little more useful:

```
<html><body>
<form action="Report" method="get">
<h2 tal:content="template/title_or_id">Title</h2>
Enter query parameters:<br><table>
<tr><th>Author</th>
<td>
  <select name="author:list" size="6" multiple>
    <option
      tal:repeat="item python:here.NewsCatalog.uniqueValuesFor('author')"
      tal:content="item"
      value="opt value">
    </option>
  </select>
</td></tr>
<tr><th>Content</th>
<td><input name="content_index" width=30 value=""></td></tr>
<tr><th>Date</th>
<td><input name="date_index" width=30 value=""></td></tr>
<tr><td colspan=2 align=center>
<input type="SUBMIT" name="SUBMIT" value="Submit Query">
</td></tr>
</table>
</form>
</body></html>
```

The new, important bit of code added to the search form is:

```
<select name="author:list" size="6" multiple>
  <option
    tal:repeat="item python:here.NewsCatalog.uniqueValuesFor('author')"
    tal:content="item"
    value="opt value">
  </option>
</select>
```

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`. When the form gets submitted, the select box will contain the exact value of an authors name, and thus match against one or more of the news objects. Your search form should look now like the figure below.

The image shows a web form titled "SearchForm". At the top, it says "Enter query parameters:". Below this, there are three input fields: "Author" is a dropdown menu with three options: "Alyssa P. Hacker", "Bob Roberts", and "Steven J. Ghoul"; "Content" is a text input field; "Date" is a text input field. At the bottom of the form is a button labeled "Submit Query".

Figure 16-6 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 `keywords` property. The `keywords` property is a lines property that lists the relevant keywords for a given Image, for example, "Portraits", "19th Century", and "Women" for a picture of Queen Victoria.

The keywords provide a way of categorizing Images. Each Image can belong in one or more categories depending on its `keywords` 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 `keywords` property. Define a *Keyword* index with the name `keywords` 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 `keywords` 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 *FieldIndex*. Using a *FieldIndex* your portrait of Queen Victoria can only be categorized one way. Using a *KeywordIndex* it can be categorized a couple different ways.

Often you will use a small list of terms with *KeywordIndexes*. In this case you may want to use the `uniqueValuesFor` method to create a custom search form. For example here's a snippet of a Page Template that will create a multiple select box for all the values in the `keywords` index:

```
<select name="keywords:list" multiple>
  <option
    tal:repeat="item python:here.uniqueValuesFor('keywords')"
    tal:content="item">
```

```
    opt value goes here
  </option>
</select>
```

Using this search form you can provide users with a range of valid search terms. You can select as many keywords as you want and Zope will find all the Images that match one or more of your selected keywords. 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 locate these objects. For example:

```
<h2>Lizard Pictures</h2>
<p tal:repeat="item
  python:here.AnimalCatalog(pathindex='/Zoo/Lizards',
    meta_type='Image')">
  <a href="url" tal:attributes="href item/absolute_url" tal:content="item/title">
    document title
  </a>
</p>
```

This query searches a ZCatalog for all images that are located within the `/Zoo/Lizards` folder and below. It creates a link to each image. To make this work, you will have to create a `FieldIndex meta_type` and `Metadata` entries for `absolute_url` and `title`.

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.

Searching DateIndexes

DateIndexes work like FieldIndexes, but are optimised for `DateTime` values. To minimize resource usage, DateIndexes have a resolution of one minute, which is considerably lower than the resolution of `DateTime` values.

DateIndexes are used just like FieldIndexes; below in the section on "Advanced Searching with Records" we present an example of searching them.

Searching DateRangeIndexes

DateRangeIndexes are specialised for searching for ranges of `DateTime` values. An example application would be `NewsItems` which have two `DateTime` attributes `effective` and `expiration`, and which should only be published if the current date would fall somewhere in between these two date values. Like DateIndexes, DateRangeIndexes have a resolution of one minute.

Searching TopicIndexes

A `TopicIndex` is a container for so-called `FilteredSets`. A `FilteredSet` consists of an expression and a set of internal ZCatalog document identifiers that represent a pre-calculated result list for performance reasons. Instead of executing the same query on a ZCatalog multiple times it is much faster to use a `TopicIndex` instead.

Building up FilteredSets happens on the fly when objects are cataloged and uncatalogued. Every indexed object is evaluated against the expressions of every FilteredSet. An object is added to a FilteredSet if the expression with the object evaluates to 1. Uncatalogued objects are removed from the FilteredSet.

A built-in type of FilteredSet is the PythonFilteredSet - it would be possible to construct custom types though.

A PythonFilteredSet evaluates using the eval() function inside the context of the FilteredSet class. The object to be indexes must be referenced inside the expression using "o.". Below are some examples of expressions.

This would index all DTML Methods:

```
o.meta_type=='DTML Method'
```

This would index all folderish objects which have a non-empty title:

```
o.isPrincipiaFolderish and o.title
```

Querying of TopicIndexes is done much in the same way as with other Indexes. Eg., if we named the last FilteredSet above `folders_with_titles`, we could query our TopicIndex with a Python snippet like:

```
zcat = context.AnimalCatalog
results = zcat(topicindex='folders_with_titles')
```

Provided our `AnimalCatalog` contains a TopicIndex `topicindex`, this would return all folderish objects in `AnimalCatalog` which had a non-empty title.

TopicIndexes also support the `operator` parameter with Records. More on Records below.

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=ZCatalog(categories=['big', 'shiny'])

# big and shiny
results=ZCatalog(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.

FieldIndex 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 passed in the `query` parameter.

For example, here is a PythonScript snippet using a range search:

```
# animals with population count greater than 5
zcat = context.AnimalCatalog
results=zcat(population_count={
    'query' : 5,
    'range': 'min'})
)
```

This query matches all objects in the AnimalCatalog which have a population count greater than 5 (provided that there is a FieldIndex `population_count` and an attribute `population_count` present).

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. Level defaults to 0, which means searching from the root. A level of -1 means start from anywhere in the path.

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** — This gives the same behaviour as our previous examples, ie. searching absolute from the root, and results in:

- /aa/bb/aa
- /aa/bb/bb
- /aa/bb/cc

query="/bb/bb", **level=0** — Again, this returns the default:

- /bb/bb/aa
- /bb/bb/bb
- /bb/bb/cc

query="/bb/bb", **level=1** — This searches for all objects which have /bb/bb one level down from the root:

- /aa/bb/bb
- /bb/bb/bb
- /cc/bb/bb

query="/bb/bb", **level=-1** — Gives all objects which have /bb/bb anywhere in their path:

- /aa/bb/bb
- /bb/bb/aa
- /bb/bb/bb
- /bb/bb/cc
- /cc/bb/bb

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) .

DateIndex Record Attributes

The supported Record Attributes are the same as those of the FieldIndex:

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 DateIndex (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 passed in the `query` parameter.

As an example, we go back to the NewsItems we created in the Section *Searching with Forms* . For this example, we created news items with attributes `content` , `author` , and `date` . Additionally, we created a search form and a report template for viewing search results.

Searching for dates of NewsItems was not very comfortable though - we had to type in exact dates to match a document.

With a `range` query we are now able to search for ranges of dates. Take a look at this PythonScript snippet:

```
# return NewsItems newer than a week
zcat = context.NewsCatalog
results = zcat( date={'query' : ZopeTime() - 7,
                    'range' : 'min'
                  })
```

DateRangeIndex Record Attributes

DateRangeIndexes only support the `query` attribute on Record objects. The `query` attribute results in the same functionality as querying directly.

TopicIndex Record Attributes

Like KeywordIndexes, TopicIndexes support the `operator` attribute:

operator — Specifies whether all FieldSets or only one need to match. Allowed values: `and` , `or` . (optional, default: `or`)

ZCTextIndex Record Attributes

Because ZCTextIndex operators are embedded in the query string, there are no additional Record Attributes for ZCTextIndexes.

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 14, Advanced Zope Scripting.

Automatic Cataloging

Automatic Cataloging is an advanced ZCatalog 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 ZCatalog 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 that people 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 ZCatalog 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 *NewsItem* . This way, when you want to add a new *NewsItem* 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 22, "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 *NewsItem* and click "Generate". This will take you back to the Products Management view and you will see your new Product.

Select the *NewsItem* 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 the figure below.

The screenshot shows the 'Add ZClass' dialog box. At the top, there's a title bar 'Add ZClass' with a 'Help!' link. Below it are three text input fields: 'Id' containing 'NewsItem', 'Title' (empty), and 'Meta Type' containing 'News Item'. A checkbox labeled 'Create constructor objects?' is checked. Below this is a section titled 'Base Classes' which is split into two panes: 'Unselected' on the left and 'Selected' on the right. The 'Unselected' pane lists several classes: 'AccessControl: User', 'AccessControl: UserFolder', 'OFS: DTMLDocument', 'OFS: DTMLMethod', 'OFS: Folder', 'OFS: File', 'OFS: Image', 'ZCatalog: CatalogAware', and 'ZCatalog: ZCatalog'. The 'Selected' pane contains 'ZCatalog: CatalogPathAware'. Between the panes are two buttons: '>>' and '<<'. At the bottom of the dialog, there is a checked checkbox 'Include standard Zope persistent object base classes?' and an 'Add' button.

Figure 16-7 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:CatalogPathAware` 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:CatalogPathAware` to show up in the right hand window. Note that if you are inheriting from more than one base class, `CatalogPathAware` should be the first (specifically, it should come before `ObjectManager`).

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 *NewsItem* 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 the figure below.

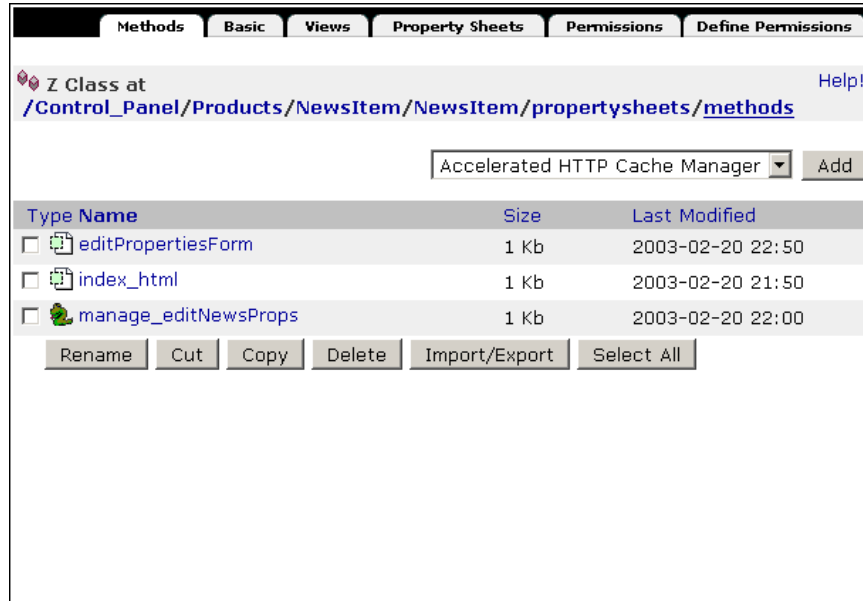


Figure 16-8 A ZClass Methods View

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 Page Templates 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.

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 the figure below.

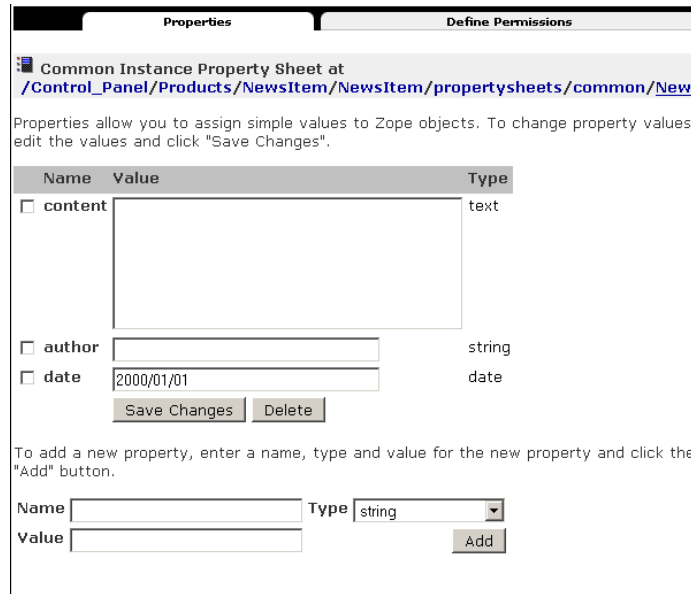


Figure 16-9 The properties screen for a Property Sheet

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 a new Form/Action pair to change the data and assigning it to a new *View* for your object.

The Form/Action pair will give you the ability to edit the data defined in the propertysheet, while the View binds the form to a tab of the Zope Management Interface.

Propertysheets come with built-in forms for editing their data; however we need to build our own so we can signal changes to the ZCatalog.

First we are going to create a form to display and edit properties. Click on the *Methods* tab. Select "Page Template" from the add drop-down menu, name it `editPropertiesForm` and fill it with:

```
<html><head>
<title tal:content="here/title_or_id">title</title>
<link rel="stylesheet" type="text/css" href="/manage_page_style.css">
</head>
<body bgcolor="#FFFFFF" link="#000099" vlink="#555555">
<span
  tal:define="manage_tabs_message options/manage_tabs_message | nothing"
  tal:replace="structure here/manage_tabs">
  prefab management tabs
</span>
```

The Zope Book (2.6 Edition)

```
<form action="manage_editNewsProps" method="get">
<table>
<tr>
  <th valign="top">Content</th>
  <td>
    <textarea
      name="content:text" rows="6" cols="35"
      tal:content="here/content">content text</textarea>
  </td>
</tr>
<tr>
  <th>Author</th>
  <td>
    <input name="author:string"
      value="author string"
      tal:attributes="value here/author">
  </td>
</tr>
<tr>
  <th>Date</th>
  <td>
    <input name="date:date"
      value="the date"
      tal:attributes="value here/date">
  </td>
</tr>
<tr><td></td><td>
<input type="submit">
</td></tr>
</form>
</body>
</html>
```

This is the Form part of the Form/Action pair. Note the call of `manage_tabs` at the top of the form - this will give your form the standard ZMI tabs.

We will add the Action part now. Add a `Script (Python)` object and fill in the id `manage_editNewsProps` and the following code:

```
# first get the request
req = context.REQUEST
# change the properties in the zclass' property sheet
context.property sheets.News.manage_editProperties(req)
# signal the change to the zcatalog
context.reindex_object()
# now return a message
form = context.editPropertiesForm
return form(REQUEST=req,
            manage_tabs_message="Saved changes.",
            )
```

Done. The next step will be to define the View. Click on the *Views* tab. This will take you to 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 "editPropertiesForm" from the select box and click *Add*. This will create a new View on this screen under the original three Views, as shown in the figure below.

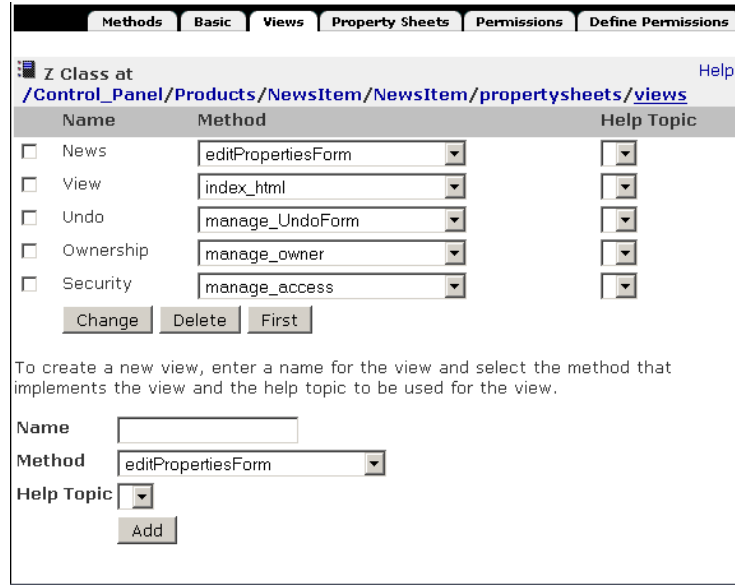


Figure 16-10 The Views view

We want to make our View 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 a method for displaying the class. Click on the *Methods* tab, select *Page Template* from the add list and add a new *Page Template* with the id "index_html". This will be the default view of your news item. Add the following to the new template:

```
<html><head>
<title tal:content="template/title">The title</title>
</head><body>
<h1>News Flash</h1>
<p tal:content="here/date">
  date goes here
</p>
<p tal:content="here/author">
  author goes here
</p>
<p tal:content="here/content">
  content goes here
</p>
</body></html>
```

Finally, we will add a new management tab for the display method. Once again, click on the *Views* tab, and create a *View* named "View", and assign the *index_html* to it. Reorder the views so that the *News* view comes first, followed by the *View* method.

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 ZCatalog that will catalog your new News Items. Go to the root folder and create a new ZCatalog with the id *Catalog*. The ZClass finds the ZCatalog by looking for a catalog named *Catalog* through acquisition, so this ZCatalog should be where it can be acquired by all NewsItems you plan to create.

Like the previous two examples of using a ZCatalog, you need to create Indexes and a Metadata Table that make sense for your objects. Create the following indexes:

content — This should be a ZCTextIndex. This will index the content of your News Items.

author — This should be a FieldIndex. This will index the author of the News Item.

date — This should be a DateIndex. This will index the date of the News Item.

After creating these Indexes, add these Metadata columns:

- author
- date
- absolute_url

After creating the Indexes and Metadata Table columns, the automatic cataloguing is basically working. The last step is creating a search interface for the ZCatalog 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 five 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 *CatalogPathAware*, 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.

Relational Database Connectivity

The Zope Object Database (ZODB) is used to store all the pages, files and other objects you create. It is fast and requires almost no setting up or maintenance. Like a filesystem, it is especially good at storing moderately-sized binary objects such as graphics.

Relational Databases work in a very different way. They are based on tables of data such as this:

Row	First Name	Last Name	Age
1	Bob	McBob	42
2	John	Johnson	24
3	Steve	Smith	38

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. This chapter assumes a basic knowledge of SQL, if you do not know SQL there are many books and tutorials on the web.

Relational databases and object databases are very different and each possesses its own strengths and weaknesses. Zope allows you to use either, providing the flexibility to choose the storage mechanism which is best for your data. The most common reasons to use relational databases are to access an existing database or to share data with other applications. Most programming languages and thousands of software products work with relational databases. Although it is possible to access the ZODB from other applications and languages, it will often require more effort than using a relational database.

By using your relational data with Zope you retain all of Zope's benefits including security, dynamic presentation, and networking. You can use Zope to dynamically tailor your data access, data presentation and data management.

Common Relational Databases

There are many relational database systems. The following is a brief list of some of the more popular database systems:

Oracle — Oracle is arguably the most powerful and popular commercial relational database. It is, however, relatively expensive and complex. Oracle can be purchased or evaluated from the Oracle Website .

DB2 — DB2 from IBM is the main commercial competitor to Oracle. It has similar power but also similar expense and complexity. More information from <http://www.ibm.com/software/data/db2/>

PostgreSQL — PostgreSQL is a leading open source relational database with good support for SQL standards. 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 .

SAP DB — An open source database developed by SAP. Has an Oracle 7 compatibility mode. More information and downloads from <http://www.sapdb.org/> .

Sybase — Sybase is another popular commercial relational database. Sybase can be purchased or evaluated from the Sybase Website .

SQL Sever — Microsoft's full featured SQL Server for the Windows operating systems. For any serious use on Windows, it is preferable to Microsoft Access. Information from <http://www.microsoft.com/sql/>

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 its entire data set into memory. You can find out more about Gadfly at <http://gadfly.sourceforge.net/> .

The mechanics of setting up relational database is different for each database and is thus 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.

Zope can connect to all the above-listed database systems; however, you should be satisfied that the database is running and operating in a satisfactory way on its own before attempting to connect it to Zope. An exception to this policy is Gadfly, which is included with Zope and requires no setup.

Database Adapters

A database can only be used if a Zope Database Adapter is available, though a Database Adapter is fairly easy to write if the database has Python support. Database adapters can be downloaded from the Products section of Zope.org The exception to this is Gadfly, which is included with Zope.

At the time of writing the following adapters were available, but this list constantly changes as more adapters are added.

Oracle — DCOracle2 package from Zope Corporation includes the ZoracleDA

DB2 — ZDB2DA from Blue Dynamics

PostgreSQL — The newest DA is ZPycopgDA included in pycopg package. The older ZpopyDA is also available.

MySQL — ZMySQLDA Available as source and a Linux binary package.

SAP DB — ZsapdbDA by Ulrich Eck.

Sybase — SybaseDA is written and commercially supported by Zope Corporation.

SQL Server — ZODBC DA is written and commercially supported by Zope Corporation. Available for the Windows platform only. This DA will also support any other ODBC compliant database.

Interbase/Firebird — There are a number of DAs available including isectZope , kinterbasdbDA and gvibDA

Informix — ZinformixDA which requires the informixdb product.

Gadfly — The Gadfly Database Adapter is built into Zope.

If you will need to connect to more than one database or wish to connect as to the same database as different users then you may use multiple database connection objects.

Setting up a Database Connection

Once the database adapter has been downloaded and installed you may create a new *Database Connection* from the *Add* menu on the Zope management pages. All database connection management interfaces are fairly similar.

The database connection object is used to establish and manage the connection to the database. Because the database runs externally to Zope, they may require you to specify information necessary to connect successfully to the database. This specification, called a *connection string*, is different for each kind of database. For example, the figure below shows the PostgreSQL database connection add form.

The screenshot shows the Zope management interface. At the top, it says 'ZOPE' and 'Logged in as admin'. The main content area is titled 'Add Z Psycog Database Connection'. On the left, there is a navigation tree with folders like 'Control_Panel', 'Examples', 'acl_users', 'temp_folder', and 'www'. The main form has the following fields and options:

- Id:** A text input field containing 'Psycog_database_connection'.
- Title:** A text input field containing 'Z Psycog Database Connection'.
- Enter a Database Connection String ¹:** A text input field containing 'dbname=test user=andrew'.
- Connect immediately:** A checked checkbox.
- Use Zope's internal DateTime module (instead of mxDateTime):** A checked checkbox.
- PyGreSQL emulation mode:** An unchecked checkbox.
- Add:** A button at the bottom of the form.

Footnote ¹ Connection Strings: The connection string used for Z Psycog Database Connection are exactly the same connection strings required by postgresql tools. The connection strings are typically of the form: dbname=database_name user=user_name password=secret_string host=server_addr port=port_number OR: dbname=database_name user=user_name password=secret_string port=port_number to use the unix socket named port_number. See postgresql documentation for more options.

Figure 17-1 PostgreSQL Database Connection

We'll be using the Gadfly database for the examples in this chapter, as it requires the least amount of configuration. If you happen to be using a different database while "playing along", note that Database Connections work slightly differently depending on which database is being used, however most have a "Test" tab for issuing a test SQL query to the database and a "Browse" tab which will show the table structure. It is good practice to use these tabs to test the database connection before going any further.

Select the *Z Gadfly Database Connection* from the add list. This will take you to the add form for a Gadfly database connection. Select and add a Gadfly connection to Zope. Note that because Gadfly runs inside Zope you do not need to specify a "connection string".

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 it exposes a button to connect or disconnect from the database. In general Zope will manage the connection to your database for you, so in practice there is little reason to manually control the connection. For Gadfly, the action of connecting and disconnecting is 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. The figure below shows the *Properties* view.

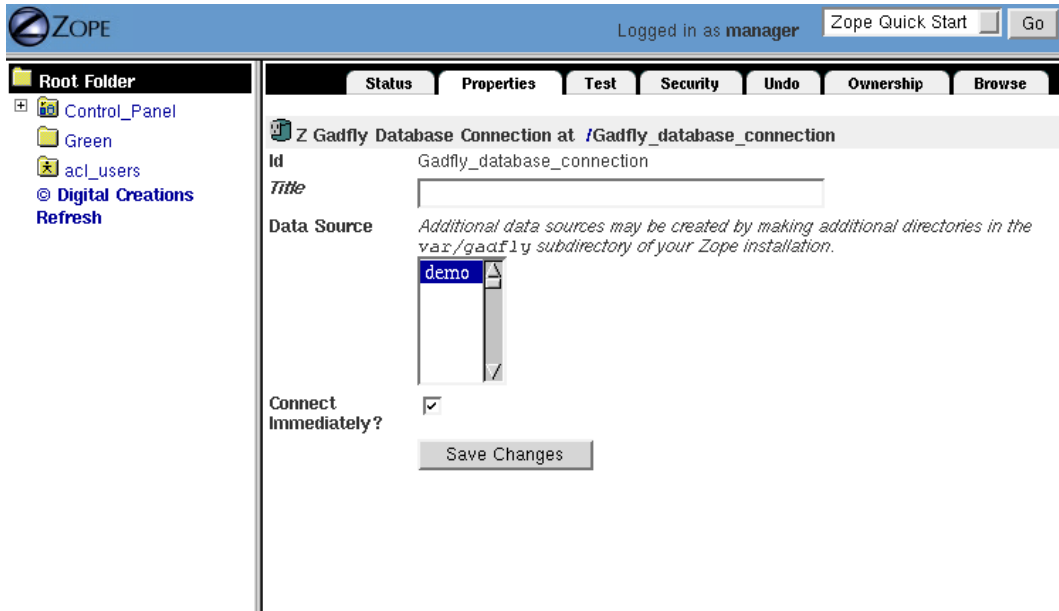


Figure 17-2 The Properties view

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 used for testing your database and issuing "one-time" SQL commands (like statements for creating tables). This is *not* the place where you will enter most of your SQL code. SQL commands typically reside in *Z SQL Methods* which will be discussed in detail later in this chapter.

Let's create a table in your database for 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 by entering it into the *Test* tab:

```
CREATE TABLE employees
(
  emp_id integer,
  first varchar,
  last varchar,
  salary float
)
```

Click the *Submit Query* button of the *Test* tab to run the SQL command. Zope should return a confirmation screen that confirms that the SQL code was run. It will additionally display the results, if any.

The SQL used here works under Gadfly but 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 you will need to consult your documentation to find out what kind of types your database supports.

To examine your table, go to the *Browse* view. This lets you view your database's tables and the schema of each table. 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 10-3.

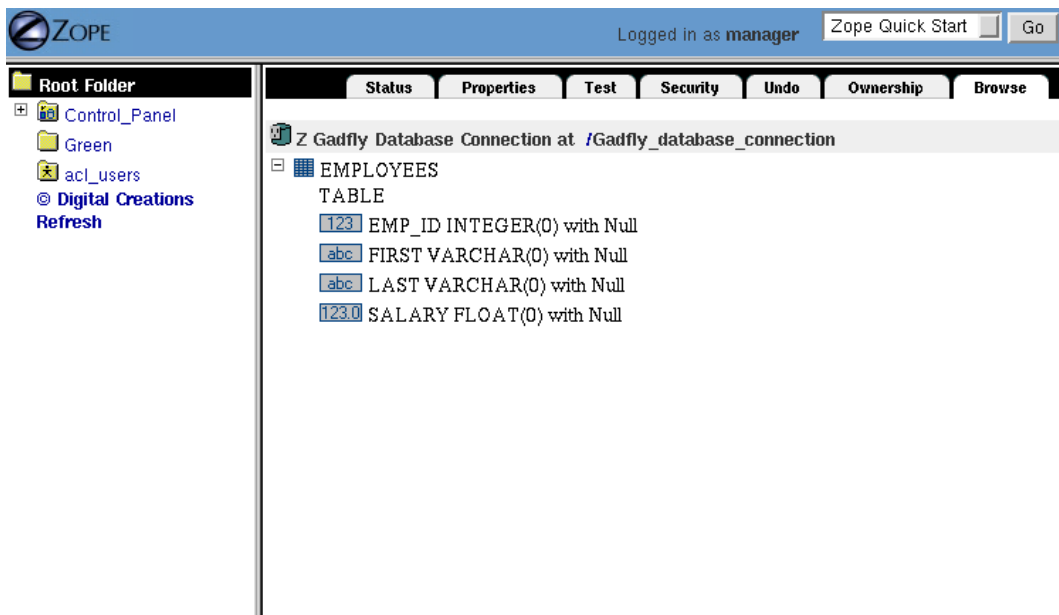


Figure 17-3 Browsing the Database Connection

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. However, 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.

Z SQL Methods

Z SQL Methods are Zope objects 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 and change database data. Z SQL Methods can also contain more than one SQL command.

A ZSQL Method has two functions: it generates SQL to send to the database and it converts the response from the database into an object. This has the following benefits:

- Generated SQL will take care of special characters that may need to be quoted or removed from the query. This speeds up code development.
- If the underlying database is changed (for example, from Postgres to Oracle), then the generated SQL will, in some cases, automatically change too, making the application more portable.
- Results from the query are packaged into an easy to use object which will make display or processing of the response very simple.
- Transactions are mediated. Transactions are discussed in more detail later in this chapter.

Examples of ZSQL Methods

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 the figure below.

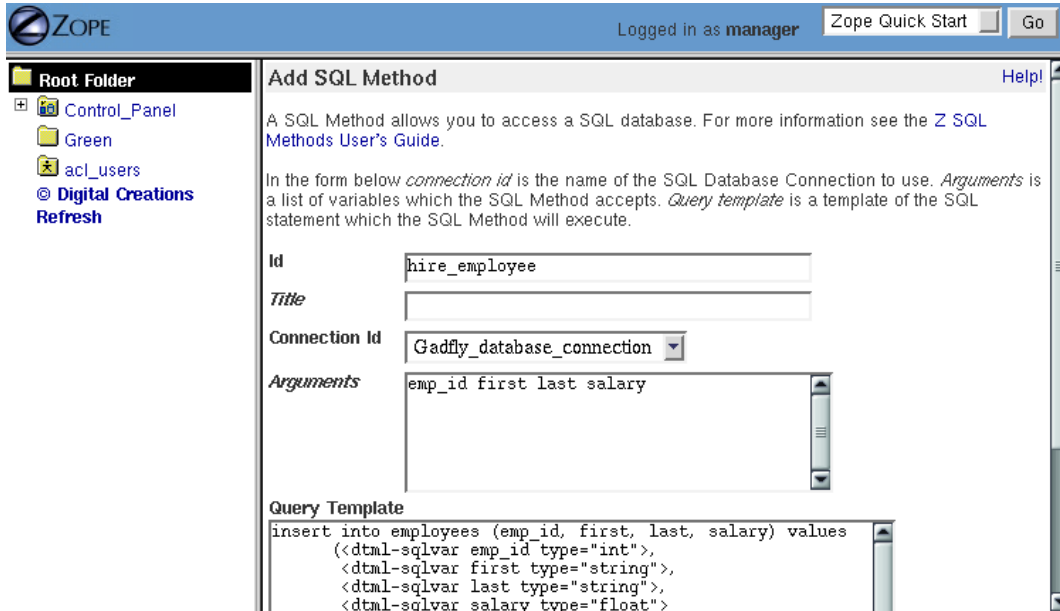


Figure 17-4 The Add form for Z SQL Methods

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, *emp_id=100* gives the *emp_id* 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. 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 of this chapter.

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 following 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 the figure below.

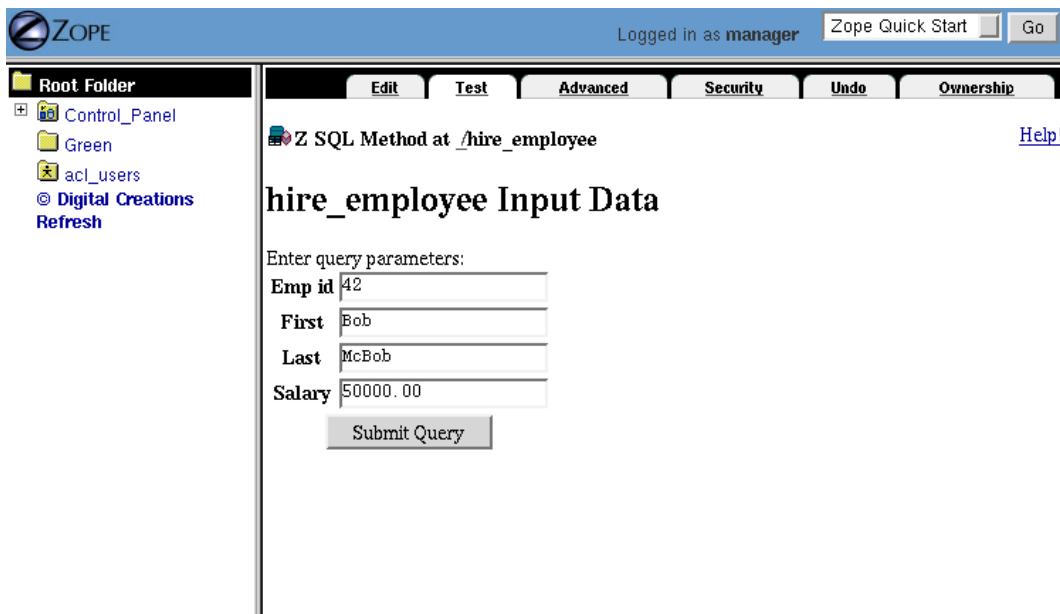


Figure 17-5 The hire_employee Test view

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 *Submit Query* 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.

Displaying Results from 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.

Somewhat predictably, as Zope is object oriented, a Z SQL method returns a *Result object*. All the result rows are packaged up into one object. For all practical purposes, the result object can be thought of as rows in the database table that have been turned into Zope objects. These objects have attributes that match the schema of the database result.

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* .

Zope Database Adapters behave slightly differently regarding how they handle different types of data. However the more modern ones will return the Python type that is closest to the SQL type - as there are far more types in SQL than in Python there cannot be a complete match. For example, a date will usually be returned as a Zope DateTime object; char, varchar and text will all be returned as strings.

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.

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 *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 the chapter entitled 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 "hireEmployeeReport" for the *Report Id* , "hireEmployeeForm" for the *Search Id* and check the "Generate DTML Methods" button then 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 *hireEmployeeReport* DTML Method a little to make it more informative. Select the *hireEmployeeReport* 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 generated *hireEmployeeReport* 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.

You now 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. Your 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 connectivity 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 REQUEST. 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 were 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 the below figure, which shows a collection of Folders in a organization's web site.

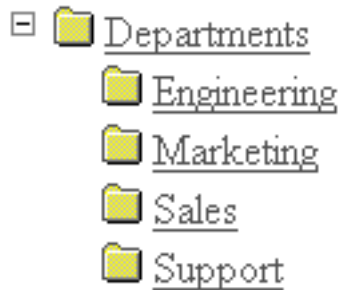


Figure 17-6 Folder structure of an organizational web site

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 the chapter entitled *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 obtaining 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*, with *emp_id* in the *Arguments* field and the following in the 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 as if you queried the ZSQL Method from DTML and passed it a single argument it would return a list of results that happened 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.

Other Result Object Methods

Up to now we have just been iterating through the attributes of the Result object in DTML. The result object does however provide other methods which can be easier in some situations. These methods can be accessed from Python scripts, page templates and from DTML. For example in Python we could write:

```
result=context.list_all_employees()
return len(result)
```

which in DTML would be:

```
<dtml-var "%.len(list_all_employees())">
```

Assuming that we have set `result` to being a result object we can use the following methods:

`len(result)` — this will show the number rows returned (which would be 3 in the example above).

`result.names()` — a list of all the column headings, returning a list containing `emp_id`, `first`, `last` and `salary`

`result.tuples()` — returns a list of tuples in our example:

```
[(43, 'Bob', 'Roberts', 50000),
 (101, 'Cheeta', 'leCat', 100000),
 (99, 'Jane', 'Junglewoman', 100001)]
```

`result.dictionaries()` — will return a list of dictionaries, with one dictionary for each row:

```
[{'emp_id': 42, 'first': 'Bob', 'last': 'Roberts', 'salary': 50000},
 {'emp_id': 101, 'first': 'Cheeta', 'last': 'leCat', 'salary': 100000},
 {'emp_id': 99, 'first': 'Jane', 'last': 'Junglewoman', 'salary': 100001}]
```

`result.data_dictionary()` — returns a dictionary describing the structure of the results table. The dictionary has the key name, type, null and width. Name and type are self explanatory, null is true if that field may contain a null value and width is the width in characters of the field. Note that null and width may not be set by some Database Adapters.

`result.asRDB()` — displays the result in a similar way to a relational database. The DTML below displays the result below:

```
<pre>
  <dtml-var "list_all_employees().asRDB()">
</pre>
```

... displays ...

```
emp_id first last salary
42 Bob Roberts 50000
101 Cheeta leCat 100000
99 Jane Junglewoman 100001
```

`result[0][1]` — return row 0, column 1 of the result, bob in this example. Be careful using this method as changes in the schema will cause unexpected results.

Binding Classes to Result Objects

A Result object has an attribute for each column in a results row. As we have seen there are some basic methods for processing these attributes to produce some more useful output. However we can go further by writing our own custom methods and adding them into the Result object.

There are two ways to bind a method to a Result object. As you saw previously, 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

binding mechanism described in the chapter entitled *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 *viewEmployee* 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. With a bit of Python, you could create brains classes that accessed network resources, called other Z SQL Methods, or performed all kinds of business logic. Since advanced Python programming is not within the scope of this book, we regrettably cannot provide a great number of examples of this sort of functionality, but we will at least provide one below.

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 the figure below.

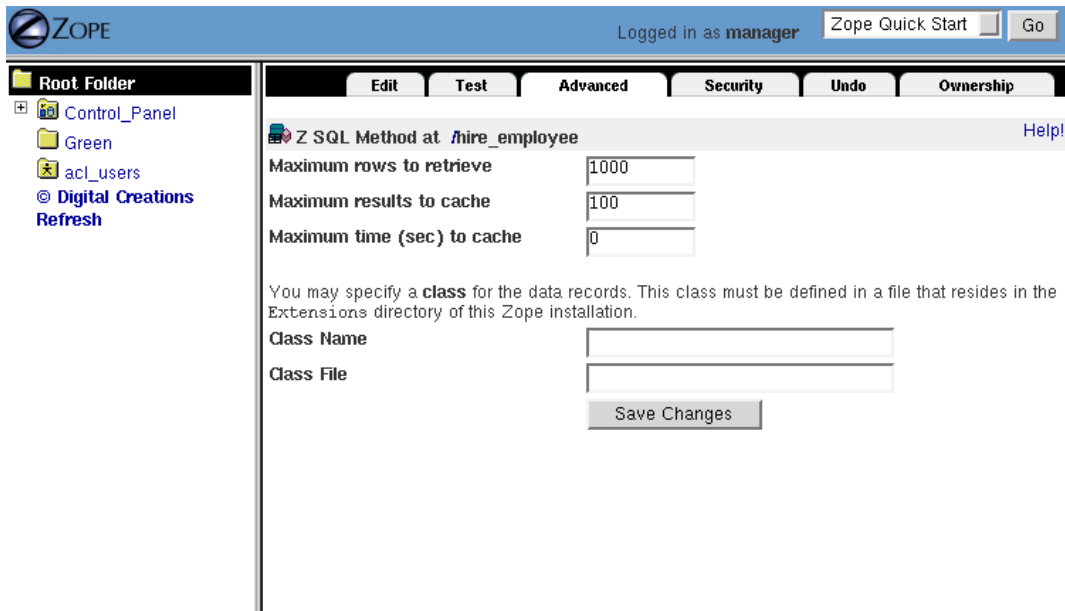


Figure 17-7 Caching controls for Z SQL Methods

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 was mentioned in the chapter entitled Zope Concepts and Architecture , 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 operation 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.

In our example, the transaction would start with the customer submitting the form from the web page and would end when the page is displayed. It is guaranteed that operations in this transaction are either all performed or none are

performed even if these operations use a mix of Zope Object Database and external relational database.

Further help

The `zope-db@zope.org` is the place to ask questions about relational databases. You can subscribe or browse the archive of previous postings at <http://lists.zope.org/mailman/listinfo/zope-db>

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 you can even use advanced Zope features like URL traversal, acquisition, undo and security while working with relational data.

Virtual Hosting Services

Zope comes with two objects that help you do virtual hosting, *SiteRoot* and *Virtual Host Monster*. Virtual hosting is a way to serve many web sites with one Zope server.

*SiteRoot*s are an artifact of an older generation of Zope virtual hosting services that are only retained in current Zope versions for backwards-compatibility purposes. They are not documented in this book because they are somewhat "dangerous" for new users, as they have the capability of temporarily "locking you out" of your Zope instance if you configure them improperly. Luckily, we have *Virtual Host Monsters*, which do everything that *SiteRoots* do and more without any of the dangerous side effects of *SiteRoots*. If you want to do virtual hosting in Zope, you should almost certainly be using a *Virtual Host Monster*.

Virtual Host Monster

Zope objects need to generate their own URLs from time to time. For instance, when a Zope object has its "absolute_url" method called, it needs to return a URL which is appropriate for itself. This URL typically contains a hostname, a port, and a path. In a "default" Zope installation, this hostname, port, and path is typically what you want. But when it comes time to serve multiple websites out of a single Zope instance, each with their own "top-level" domain name, or when it comes time to integrate a Zope Folder within an existing website using Apache or another webserver, the URLs that Zope objects generate need to change to suit your configuration.

A Virtual Host Monster's only job is to change the URLs which your Zope objects generate. This allows you to customize the URLs that are displayed within your Zope application, allowing an object to have a different URL when accessed in a different way. This is most typically useful, for example, when you wish to "publish" the contents of a single Zope Folder (e.g. /FooFolder) as a URL that does not actually contain this Folder's name (e.g as the hostname `http://www.foofolder.com/`).

The Virtual Host Monster performs this job by intercepting and deciphering information passed to Zope within special path elements encoded in the URLs of requests which come in to Zope. If these special path elements are absent in the URLs of requests to the Zope server, the Virtual Host Monster does nothing. If they are present, however, the Virtual Host Monster decipheres the information passed in via these path elements and causes your Zope objects to generate a URL that is different from their "default" URL.

The Zope values which are effected by the presence of a Virtual Host Monster include REQUEST variables starting with URL or BASE (such as URL1, BASE2, URLPATH0), and the absolute_url() methods of objects.

Virtual Host Monster configuration can be complicated, because it requires that you *rewrite* URLs "on the way in" to Zope. In order for the special path elements to be introduced into the URL of the request sent to Zope, a front-end URL "rewriting" tool needs to be employed. Virtual Host Monster comes with a simple rewriting tool in the form of its *Mappings* view, or alternately you can use Apache or another webserver to rewrite URLs of requests destined to Zope for you.

Where to Put a Virtual Host Monster And What To Name It

A single Virtual Host Monster in your Zope root can handle all of your virtual hosting needs. It doesn't matter what id you give it, as long as nothing else in your site has the same id .

Special VHM Path Elements *VirtualHostBase* and *VirtualHostRoot*

A Virtual Host Monster doesn't do anything unless it sees one of the following special path elements in a URL:

VirtualHostBase — if a VirtualHostMonster "sees" this name in the incoming URL, it causes Zope objects to generate URLs with a potentially different protocol, a potentially different hostname, and a potentially different port number.

VirtualHostRoot — if a VirtualHostMonster "sees" this name in the incoming URL, it causes Zope objects to generate URLs which have a potentially different "path root"

VirtualHostBase

The `VirtualHostBase` declaration is typically found at the beginning of an incoming URL. A Virtual Host Monster will intercept two path elements following this name and will use them to compose a new protocol, hostname, and port number.

The two path elements which must follow a `VirtualHostBase` declaration are `protocol` and `hostname:portnumber`. They must be separated by a single slash. The colon and `portnumber` parts of the second element are optional, and if they don't exist, the Virtual Host Monster will not change the port number of Zope-generated URLs.

Examples:

If a VHM is installed in the root folder, and a request comes in to your Zope with the URL:

```
'http://zopeserver:8080/VirtualHostBase/http/www.buystuff.com'
```

URLs generated by Zope objects will start with
'http://buystuff.com:8080'.

If a VHM is installed in the root folder, and a request comes in to your Zope with the URL:

```
'http://zopeserver:8080/VirtualHostBase/http/www.buystuff.com:80'
```

URLs generated by Zope objects will start with
'http://buystuff.com' (port 80 is the default port number so it is left out).

If a VHM is installed in the root folder, and a request comes in to your Zope with the URL:

```
'http://zopeserver:8080/VirtualHostBase/https/www.buystuff.com:443'
```

URLs generated by Zope objects will start with
'https://buystuff.com/'. (port 443 is the default https port number, so it is left off).

One thing to note when reading the examples above is that if your Zope is running on a port number like 8080, and you want generated URLs to not include this port number and instead be served on the standard HTTP port (80), you must specifically include the default port 80 within the `VirtualHostBase` declaration, e.g.

`/VirtualHostBase/http/www.buystuff.com:80`. If you don't specify the `:80`, your Zope's HTTP port number will be used (which is likely not what you want).

VirtualHostRoot

The `VirtualHostRoot` declaration is typically found near the end of an incoming URL. A Virtual Host Monster will gather up all path elements which *precede* and *follow* the `VirtualHostRoot` name, traverse the Zope object hierarchy with these elements, and publish the object it finds with the path rewritten to the path element(s) which *follow* (s) the `VirtualHostRoot` name.

This is easier to understand by example. For a URL `/a/b/c/VirtualHostRoot/d`, the Virtual Host Monster will traverse "a/b/c/d" and then generate a URL with path `/d`.

Examples:

If a VHM is installed in the root folder, and a request comes in to your Zope with the URL:

```
'http://zopeserver:8080/Folder/VirtualHostRoot/
```

The object 'Folder' will be traversed to and published, URLs generated by Zope will start with 'http://zopeserver:8080/', and when they are visited, they will be considered relative to 'Folder'.

If a VHM is installed in the root folder, and a request comes in to your Zope with the URL:

```
'http://zopeserver:8080/HomeFolder/VirtualHostRoot/Chris
```

The object '/Folder/Chris' will be traversed to and published, URLs generated by Zope will start with 'http://zopeserver:8080/Chris', and when they are visited, they will be considered relative to '/HomeFolder/Chris'.

Using `VirtualHostRoot` and `VirtualHostBase` Together

The most common sort of virtual hosting setup is one in which you create a Folder in your Zope root for each domain that you want to serve. For instance the site `http://www.buystuff.com` is served from a Folder in the Zope root named `/buystuff` while the site `http://www.mycause.org` is served from a Folder in the Zope root named `/mycause`. In order to do this, you need to generate URLs that have both `VirtualHostBase` and `VirtualHostRoot` in them.

To access `/mycause` as `http://www.mycause.org/`, you would cause Zope to be visited via the following URL:

```
/VirtualHostBase/http/www.mycause.org:80/mycause/VirtualHostRoot/
```

In the same Zope instance, to access `/buystuff` as `http://www.buystuff.com/`, you would cause Zope to be visited via the following URL:

```
/VirtualHostBase/http/www.buystuff.com:80/buystuff/VirtualHostRoot/
```

Testing a Virtual Host Monster

Set up a Zope on your local machine that listens on HTTP port 8080 for incoming requests.

Visit the root folder, and select *Virtual Host Monster* from the Add list. Fill in the `id` on the add form as `VHM` and click Add.

Create a Folder in your Zope root named `vhm_test`. Within the newly-created `vhm_test` folder, create a DTML Method named `index_html` and enter the following into its body:

```
<html>
<body>
<table border="1">
  <tr>
    <td>Absolute URL</td>
    <td><dtml-var absolute_url></td>
  </tr>
  <tr>
    <td>URL0</td>
    <td><dtml-var URL0></td>
  </tr>
</table>
```

```
<tr>
  <td>URL1</td>
  <td><dtml-var URL1></td>
</tr>
</table>
</body>
</html>
```

View the DTML Method by clicking on its View tab, and you will see something like the following:

```
Absolute URL  http://localhost:8080/vhm_test
URL0         http://localhost:8080/vhm_test/index_html
URL1         http://localhost:8080/vhm_test
```

Now visit the URL `http://localhost:8080/vhm_test` . You will be presented with something that looks almost exactly the same.

Now visit the URL `http://localhost:8080/VirtualHostBase/http/zope.com:80/vhm_test` . You will be presented with something that looks much like this:

```
Absolute URL  http://zope.com/vhm_test
URL0         http://zope.com/vhm_test/index_html
URL1         http://zope.com/vhm_test
```

Note that the URLs that Zope is generating have changed. Instead of using `localhost:8080` for the hostname and path, we've instructed Zope, through the use of a `VirtualHostBase` directive to use `zope.com` as the hostname. No port is shown because we've told Zope that we want to generate URLs with a port number of 80, which is the default http port.

Now visit the URL

`http://localhost:8080/VirtualHostBase/http/zope.com:80/vhm_test/VirtualHostRoot/` . You will be presented with something that looks much like this:

```
Absolute URL  http://zope.com
URL0         http://zope.com/index_html
URL1         http://zope.com
```

Note that we're now publishing the `vhm_test` folder as if it were the root folder of a domain named `zope.com` . We did this by appending a `VirtualHostRoot` directive to the incoming URL, which essentially says "traverse to the `vhm_root` folder as if it were the root of the site."

Arranging for Incoming URLs to be Rewritten

At this point, you're probably wondering just how in the world any of this helps you. You're certainly not going to ask people to use their browser to visit a URL like

`http://yourserver.com/VirtualHostBase/http/zope.com/vhm_test/VirtualHostRoot/` just so your Zope-generated URLs will be "right". That would defeat the purpose of virtual hosting entirely. The answer is: don't ask humans to do it, ask your computer to do it. There are two common (but mutually exclusive) ways to accomplish this: via the `VirtualHostMonster Mappings` tab and via Apache "rewrite rules" (or your webserver's facility to do the same thing if you don't use Apache). Be warned: use either one of these facilities or the other but not both or very strange things may start to happen. We give examples of using both facilities below.

Virtual Host Monster Mappings Tab

Use the Virtual Host Monster's *Mappings* tab to cause your URLs to be rewritten if:

- You run a "bare" Zope without a front-end webserver like Apache.

- You have one or more folders in your Zope that you'd like to publish as "http://some.hostname.com/" instead of "http://hostname.com/a/folder".

The lines entered into the *Mappings* tab are in the form

The best way to explain how to use the *Mappings* tab is by example. Assuming you've added a Virtual Host Monster object in your root folder on a Zope running on `localhost` on port 8080, create an alias in your local system's `hosts` file (in `/etc/hosts` on UNIX and in `c:\WINNT\system32\drivers\etc\hosts` on Windows) that looks like this:

```
127.0.0.1 www.example.com
```

This causes your local machine to contact itself when a hostname of `www.example.com` is encountered. For the sake of this example, we're going to want to contact Zope via the hostname `www.example.com` through a browser (also on your local host) and this makes it possible.

Then visit the VHM in the root folder and click on its *Mappings* tab. On a line by itself enter the following:

```
www.example.com:8080/vhm_test
```

This will cause the `vhm_test` folder to be published when we visit `http://www.example.com:8080`. Visit `http://www.example.com:8080`. You will see:

```
Absolute URL  http://www.example.com:8081
URL0         http://www.example.com:8080/index_html
URL1         http://www.example.com:8080
```

In the "real world" this means that you are "publishing" the `vhm_test` folder as `http://www.example.com:8080`.

You can match multiple subdomains by putting "." in front of the host name in the mapping rule. For example, ".buystuff.com" will match "my.buystuff.com", "zoom.buystuff.com", etc. If an exact match exists, it is used instead of a wildcard match.

Apache Rewrite Rules

If you use Apache in front of Zope, instead of using the *Mappings* tab, you should use Apache's rewrite rule functionality to rewrite URLs in to Zope. The way this works is straightforward: Apache listens on its "normal" port, typically port 80. At the same time, Zope's web server (on the same host or on another host) listens on a different port (typically 8080). Apache accepts requests on its listening port. A virtual host declaration in Apache's configuration tells Apache to rewrite

Using Apache's rewrite rule functionality requires that the `mod_rewrite` Apache module be enabled. This is typically done by configuring Apache with the `--enable-module=rewrite` flag.

After you've got Apache configured with `mod_rewrite`, you can start configuring Apache's config file and Zope for the following example. Assuming you've added a Virtual Host Monster object in your root folder on a Zope running on `localhost` on port 8080, create an alias in your local system's `hosts` file (in `/etc/hosts` on UNIX and in `c:\WINNT\system32\drivers\etc\hosts` on Windows) that looks like this:

```
127.0.0.1 www.example.com
```

This causes your local machine to contact itself when a hostname of `www.example.com` is encountered. For the sake of this example, we're going to want to contact Zope via the hostname `www.example.com` through a browser (also on your local host) and this makes it possible.

Now, assuming you've got Apache running on port 80 and Zope running on port 8080 on your local machine, and assuming that you want to serve the folder named `vhm_test` in Zope as `www.example.com` and, add the following

to your Apache's `httpd.conf` file and restart your Apache process:

```
NameVirtualHost *
<VirtualHost *>
ServerName www.example.com
RewriteEngine On
RewriteRule ^/(.*) http://127.0.0.1:8080/VirtualHostBase/http/www.example.com:80/vhm_test/VirtualHostRoot/$1 [L,P]
</VirtualHost>
```

When you visit `http://www.example.com` in your browser, you will see:

```
Absolute URL http://www.example.com
URL0         http://www.example.com/index_html
URL1         http://www.example.com
```

This page is being served by Apache, but the results are coming from Zope. Requests come in to Apache with "normal" URLs (e.g. `http://www.example.com`). The `VirtualHost` stanza in Apache's `httpd.conf` causes the request URL to be rewritten (e.g. to `http://127.0.0.1:8080/VirtualHostBase/http/www.example.com:80/vhm_test/VirtualHostRoot/`). Apache then calls the rewritten URL, and returns the result.

See the Apache Documentation for more information on the subject of rewrite rules.

"Inside-Out" Virtual Hosting

Another use for virtual hosting is to make Zope appear to be part of a site controlled by another server. For example, Zope might only serve the contents of `http://www.mycause.org/dynamic_stuff` , while Apache or another webserver serves files via `http://www.mycause.org/` . To accomplish this, you want to add "dynamic_stuff" to the start of all Zope-generated URLs.

If you insert `VirtualHostRoot`, followed by one or more path elements that start with `_vh_` , then these elements will be ignored during traversal and then added (without the `_vh_`) to the start of generated URLs. For instance, a request for `/a/VirtualHostRoot/_vh_z/` will traverse "a" and then generate URLs that start with /z.

In our example, you would have the main server send requests for `http://www.mycause.org/dynamic_stuff/anything` to Zope, rewritten as `/VirtualHostRoot/_vh_dynamic_stuff/anything`.

Sessions

Sessions allow you to keep state between HTTP requests for site users, making implementing things like "shopping carts" and other applications which must maintain ad-hoc state related to a site visitor for some period of time greater than a single request. Zopes 2.5.0 and greater have built-in sessioning machinery, which is described in this chapter.

Introduction

Sessions allow you keep track of site visitors. Web browsers use the HTTP protocol to exchange data with Zope. HTTP does not provide a way to associate subsequent requests from the same user: each request is considered completely independent of the last.

Sessions help overcome this limitation. 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, HTTP form elements, and/or parts of URLs "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".

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 in a session unless you understand the risks involved in doing so. See the section entitled *Security Considerations* near the end of this document to become more familiar with these risks.

Additionally, it is advisable to use sessions only on pages where they are really necessary as they will have a performance impact on your application. The severity of this impact varies depending on usage and configuration, but a good "rule of thumb" is to account for a 5% - 10% speed-of-execution penalty when viewing a page which references a session versus a similar page which does not. This penalty can vary wildly depending on the number of "writes" you perform during your use of sessioning, so (as always) it is wise to test your application which uses sessions under load before putting it in to production.

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, see the *Details* section in this chapter.

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, form variables, or URL path elements or a combination of the aforementioned. 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 Script (Python) object:

```
## Script (Python) "sessionTest"
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'
```

Note that this script is very simpleminded example which demonstrates how how Zope sessions work. It is not a "best practice" example. You should probably not attempt to keep a last-accessed time in this manner in a production application because it might slow your application down dramatically and cause problems under high load.

Create a script with this body named `sessionTest` in your root folder, and run it via its `Test` tab. While viewing the test output, reload the workspace frame a few times. Note that the script keeps track of when you last viewed it 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 within 20 minutes or so, it still remembers the last time you viewed it.

This example shows the basic features of working with session data: session data objects act much like Python dictionaries. A session data object can hold almost any kind of object as a key or a value, but it's likely that you will almost always use session data that consists of "normal" Python objects such as 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.get("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
```

If you forget to perform the last step (`session["myList"]=l`) in the script, the changes to the list will not be saved, and on a subsequent request, "spam" will not show up in the `myList` list. This is a limitation of the storage mechanism used by sessions (Zope's ZODB). For more information about persistence and mutable data, see the Persistence chapter of 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 detailed below.

For an additional example of using sessions, see the "shopping cart" example that comes with Zope 2.5 and above (in the Examples folder).

Details

There are four major components to the Zope sessioning machinery design. These are described in detail below:

- **Browser Id Manager** -- this is the component which determines a remote client's "browser id", which uniquely identifies a particular browser. The browser id is encoded in a form/querystring variable, a cookie variable, or as part of the URL. The browser id manager examines cookies, form and querystring elements, and URLs to determine the client's browser id. It can also modify cookies and URLs automatically in order to differentiate users between requests. There may be more than one browser id manager in a Zope installation, but commonly there will only be one. Application developers will generally not talk directly to a browser id manager. Instead, they will use the SESSION object called out of REQUEST.SESSION, which will delegate some calls to a browser id manager. Browser id managers have "fixed" Zope ids so they can be found via acquisition by session data managers. Browser id managers also have interfaces for encoding a URL with browser id information and performing other utility functions.
- **Session Data Manager** -- this is the component which is responsible for handing out session data to callers. When session data is required, the session data manager talks to a browser id manager to determine the current browser id and creates a new session data object or hands back an existing session data object based on the browser id. Developers will generally not directly use methods of session data managers to obtain session data objects when writing application code. Instead, they will rely on the built-in REQUEST.SESSION object, which represents *the current session data object related to the user's browser id*. The session data object itself has an identifier which is different than the browser id. This identifier represents a single user session with the server (unlike the browser id, which represents a single browser). Many session data managers can use one browser id manager. Many session data managers can be instantiated on a single Zope installation. Different session data managers can implement different policies related to session data object storage (e.g. to which session data container the session data objects are stored).
- **Transient Data Container (aka Session Data Container)** -- this is the component which actually holds information related to sessions. Currently, it is used to hold a special "transient data object" (aka "session data object") instance for each ongoing session. Developers will generally not interact with transient data containers. Transient data containers are responsible for expiring the session data objects which live within them.
- **Transient Data Object (aka Session Data Object)** -- these are the objects which are stored in session data containers and managed by transient data managers. Developers interact with a transient data object after obtaining one via REQUEST.SESSION or from a session data manager directly. A single transient data object actually stores the useful information related to a single user's session. Transient data objects can be expired automatically by transient data containers as a result of inactivity, or they can be manually invalidated in the

course of a script.

Terminology

Here's a mini-glossary of terminology used by the session tracking product:

Browser Id — the string or integer used to represent a single anonymous visitor to the part of the Zope site managed by a single browser id manager. E.g. "12083789728".

Browser Id Name — the name which is looked for in places enumerated by the currently configured browser id namespaces. E.g. "_ZopeId".

Browser Id Namespaces — the browser id name will be found in one of three possible places ("namespaces"): in form elements and/or query strings (aka "form"), in a cookie, or in the URL.

Session Data Object — an transient data object that is found by asking a session data container for the item with a key that is the current browser id value.

Session Id — the identifier for a session data object. This is different than the browser id. Instead of representing a single *visitor*, it represents a single *visit*.

Default Configuration

Zope is preconfigured with a default sessioning setup as of Zope versions 2.5 and higher.

The Zope "default" browser id manager lives in the root folder and is named `browser_id_manager`.

The Zope "default" session data manager lives in the root folder and is named `session_data_manager`.

A "default" transient data container (session data container) is created as `/temp_folder/session_data` when Zope starts up. The `temp_folder` object is a "mounted, nonundoing" database that keeps information in RAM, so "out of the box", Zope stores session information in RAM. The temp folder is a "nonundoing" storage (meaning you cannot undo transactions which take place within it) because accesses to transient data containers are very write-intensive, and undoability adds unnecessary overhead.

A transient data container stores transient data objects. The default implementation the transient data object shipped with Zope is engineered to reduce the potential inherent in the ZODB for "conflict errors" related to the ZODB's "optimistic concurrency" strategy.

You needn't change any of these default options to use sessioning under Zope unless you want to customize your setup. However, if you have custom needs, can create your own session data managers, browser id managers, temporary folders, and transient object containers by choosing these items from Zope's "add" list in the place of your choosing.

Advanced Development Using Sessioning

Overview

Developers generally interact with a session data object named `REQUEST.SESSION` in order to make use of sessioning in Zope. When you work with the `REQUEST.SESSION` object, you are working with a "session data object" that is related to the current site user.

Session data objects have methods of their own, including methods which allow developers to get and set data. Session data objects are also "wrapped" in the acquisition context of their session data manager, so you may additionally call any method on a session data object that you can call on a session data manager. For information about the API of a session data manager and a session data object, see the Zope Help system item in "Zope Help" -> "API Reference" -> "Session API".

Obtaining A Session Data Object

The session data object associated with the browser id in the current request may be obtained via `REQUEST.SESSION`. If a session data object does not exist in the session data container, one will be created automatically when you reference `REQUEST.SESSION`:

```
<dtml-let data="REQUEST.SESSION">
  The 'data' name now refers to a new or existing session data object.
</dtml-let>
```

You may also use the `getSessionData()` method of a session data manager to do the same thing:

```
<dtml-let data="session_data_manager.getSessionData()">
  The 'data' name now refers to a new or existing session data object.
</dtml-let>
```

A reference to `REQUEST.SESSION` or `getSessionData()` implicitly creates a new browser id if one doesn't exist in the current request. These mechanisms also create a new session data object in the session data container if one does not exist related to the browser id in the current request. To inhibit this behavior, use the `create=0` flag to the `getSessionData()` method:

```
<dtml-let
  data="session_data_manager.getSessionData(create=0)"> The
  'data' name now refers to an existing session data object
  or None if there was no existing browser id or session data
  object. </dtml-let>
```

Modifying A Session Data Object

Once you've used `REQUEST.SESSION` or `session_data_manager.getSessionData()` to obtain a session data object, you can set key/value pairs of that session data object. These key/value pairs are where you store information related to a particular anonymous visitor. You can use the `set`, `get`, and `has_key` methods of session data objects to perform actions related to manipulating data stored in a session data object:

```
<dtml-let data="REQUEST.SESSION">
  <dtml-call "data.set('foo', 'bar')">
  <dtml-comment>Set 'foo' key to 'bar' value.</dtml-comment>
  <dtml-var "data.get('foo')">
  <dtml-comment>Will print 'bar'</dtml-comment>
  <dtml-if "data.has_key('foo')">
    This will be printed.
  <dtml-else>
    This will not be printed.
  </dtml-if>
</dtml-let>
```

An essentially arbitrary set of key/value pairs can be placed into a session data object. Keys and values can be any kinds of Python objects (note: see *Concepts and Caveats* section below wfor exceptions to this rule). The session data container which houses the session data object determines its expiration policy. Session data objects will be available across client requests for as long as they are not expired.

Manually Invalidating A Session Data Object

Developers can manually invalidate a session data object. When a session data object is invalidated, it will be flushed from the system, and will not be returned by subsequent references to `REQUEST.SESSION` or `getSessionData()`. The `invalidate()` method of a session data object causes this to happen:

```
<dtml-let data="REQUEST.SESSION">
  <dtml-call "data.invalidate()">
</dtml-let>
```

This session data object will be invalidated, and subsequent references to `REQUEST.SESSION` in this same request will return a new session data object. Manual invalidation of session data is useful in cases where you know the session data is stale and you wish to flush it from the data manager.

If an "onDelete" event is defined for a session data object, the `onDelete` method will be called before the data object is invalidated. See a following section for information about session data object "onDelete" and "onAdd" events.

Manually Invalidating A Browser Id Cookie

Invalidating a session data object does not invalidate the browser id cookie stored on the user's browser. Developers may manually invalidate the cookie associated with the browser id. To do so, they can use the `flushBrowserIdCookie()` method of a browser id manager. For example:

```
<dtml-call "REQUEST.SESSION.getBrowserIdManager().flushBrowserIdCookie()">
```

If the `cookies` namespace isn't a valid browser id key namespace when this call is performed, an exception will be raised.

An Example Of Using Session Data from DTML

An example of obtaining a session data object and setting one of its key-value pairs in DTML follows:

```
<dtml-let a="REQUEST.SESSION">
  Before change: <dtml-var a><br>
  <dtml-call "a.set('zopetime', ZopeTime())">
  <dtml-comment>
    'zopetime' will be set to a datetime object for the current
    session
  </dtml-comment>
  After change: <dtml-var a><br>
</dtml-let>
```

The first time you run this method, the "before change" representation of the session data object will be that of an empty dictionary, and the "after change" representation will show a key/value pair of `zopetime` associated with a date and time. Assuming you've configured your browser id manager with cookies and they're working on your browser properly, the second and subsequent times you view this method, the "before change" representation of the session data object will have date and time in it that was the same as the last call's "after change" representation of the same session data object. This demonstrates the very basics of session management, because it demonstrates that we are able to associate an object (the session data object obtained via `REQUEST.SESSION`) with an anonymous visitor between HTTP requests.

NOTE: To use this method in conjunction with formvar-based sessioning, you'd need to encode a link to its URL with the browser id by using the browser id manager's `encodeUrl()` method.

Using the mapping Keyword With A Session Data Object in a dtml-with

DTML has the facility to treat a session data object as a mapping, making it easier to spell some of the more common methods of access to session data objects. The `mapping` keyword to `dtml-with` means "treat name lookups that follow

this section as queries to my contents by name." For example:

```
<dtml-let a="REQUEST.SESSION">
  <dtml-call "a.set('zopetime', ZopeTime())">
  <dtml-comment>
    'zopetime' will be set to a datetime object for the current
    session... the "set" it calls is the set method of the
    session data object.
  </dtml-comment>
</dtml-let>

<dtml-with "REQUEST.SESSION" mapping>
  <dtml-var zopetime>
  <dtml-comment>
    'dtml-var zopetime' will print the DateTime object just set
    because we've used the mapping keyword to map name lookups
    into the current session data object.
  </dtml-comment>
</dtml-with>
```

Using Session Data From Python

Here's an example of using a session data manager and session data object from a set of Python external methods:

```
import time
def setCurrentTime(self):
    a = self.REQUEST.SESSION
    a.set('thetime', time.time())

def getLastTime(self):
    a = self.REQUEST.SESSION
    return a.get('thetime')
```

Calling the `setCurrentTime` will method will set the value of the current session's "thetime" key to an integer representation of the current time. Calling the `getLastTime` external method will return the integer representation of the last known value of "thetime".

Interacting with Browser Id Data

You can obtain the browser id value associated with the current request:

```
<dtml-var "REQUEST.SESSION.getBrowserIdManager().getBrowserId() ">
```

Another way of doing this, which returns the same value is:

```
<dtml-var "REQUEST.SESSION.getContainerKey() ">
```

This snippet will print the browser id value to the remote browser. If no browser id exists for the current request, a new browser id is created implicitly and returned.

If you wish to obtain the current browser id value without implicitly creating a new browser id for the current request, you can ask the `browser_id_manager` object explicitly for this value with the `create=0` parameter :

```
<dtml-var "browser_id_manager.getBrowserId(create=0) ">
```

This snippet will print a representation of the `None` value if there isn't a browser id associated with the current request, or it will print the browser id value if there is one associated with the current request. Using `create=0` is useful if you do not wish to cause the sessioning machinery to attach a new browser id to the current request, perhaps if you do not wish a browser id cookie to be set.

The browser id is either a string or an integer and has no business meaning. In your code, you should not rely on the browser id value composition, length, or type as a result, as it is subject to change.

Determining Which Namespace Holds The Browser Id

For some applications, it is advantageous to know from which namespace (currently one of "cookies", "form", or "url") the browser id has been gathered. There are three methods of browser id managers which allow you to accomplish this, `isBrowserIdFromCookie()`, `isBrowserIdFromForm()`, and `isBrowserIdFromUrl()`:

```
<dtml-if "REQUEST.SESSION.getBrowserIdManager().isBrowserIdFromCookie(">
  The browser id came from a cookie.
</dtml-if>

<dtml-if "REQUEST.SESSION.getBrowserIdManager().isBrowserIdFromForm(">
  The browser id came from a form.
</dtml-if>

<dtml-if "REQUEST.SESSION.getBrowserIdManager().isBrowserIdFromUrl(">
  The browser id came from the URL.
</dtml-if>
```

The `isBrowserIdFromCookie()` method will return true if the browser id in the current request comes from the `REQUEST.cookies` namespace. This is true if the browser id was sent to the Zope server as a cookie.

The `isBrowserIdFromForm()` method will return true if the browser id in the current request comes from the `REQUEST.form` namespace. This is true if the browser id was sent to the Zope server encoded in a query string or as part of a form element.

The `isBrowserIdFromUrl()` method will return true if the browser id in the current request comes from the leading elements of the URL.

If a browser id doesn't actually exist in the current request when one of these methods is called, an error will be raised.

During typical operations, you shouldn't need to use these methods, as you shouldn't care from which namespace the browser id was obtained. However, for highly customized applications, this set of methods may be useful.

Obtaining the Browser Id Name/Value Pair and Embedding It Into A Form

You can obtain the browser id name from a browser id manager instance. We've already determined how to obtain the browser id itself. It is useful to also obtain the browser id name if you wish to embed a browser id name/value pair as a hidden form field for use in POST requests:

```
<html>
<body>
<form action="thenextmethod">
<input type=submit name="submit" value=" GO ">
<input type=hidden name="<dtml-var "REQUEST.SESSION.getBrowserIdManager().getBrowserIdName(">
  value="<dtml-var "REQUEST.SESSION.getBrowserIdManager().getBrowserId(">">
</form>
</body>
</html>
```

A convenience function exists for performing this action as a method of a browser id manager named `getHiddenFormField()`:

```
<html>
<body>
<form action="thenextmethod">
<input type="submit" name="submit" value=" GO ">
<dtml-var "REQUEST.SESSION.getBrowserIdManager().getHiddenFormField(">
</form>
</body>
</html>
```

When either of the above DTML snippets are rendered, the resulting HTML will look something like this:

```
<html>
<body>
<form action="thenextmethod">
<input type="submit" name="submit" value=" GO ">
<input type="hidden" name="_ZopeId" value="9as09a7fs70y1j2hd7at8g">
</form>
</body>
</html>
```

Note that to maintain state across requests when using a form submission, even if you've got `Automatically Encode Zope-Generated URLs With a Browser Id` checked off in your browser id manager, you'll either need to encode the form "action" URL with a browser id (see "Embedding A Browser Id Into An HTML Link" below) or embed a hidden form field.

Determining Whether A Browser Id is "New"

A browser id is "new" if it has been set in the current request but has not yet been acknowledged by the client. "Not acknowledged by the client" means it has not been sent back by the client in a request. This is the case when a new browser id is created by the sessioning machinery due to a reference to `REQUEST.SESSION` or similar as opposed to being received by the sessioning machinery in a browser id name namespace. You can use the `isBrowserIdNew()` method of a browser id manager to determine whether the session is new:

```
<dtml-if "REQUEST.SESSION.getBrowserIdManager().isBrowserIdNew(">
  Browser id is new.
<dtml-else>
  Browser id is not new.
</dtml-if>
```

This method may be useful in cases where applications wish to prevent or detect the regeneration of new browser ids when the same client visits repeatedly without sending back a browser id in the request (such as may be the case when a visitor has cookies "turned off" in their browser and the browser id manager only uses cookies).

If there is no browser id associated with the current request, this method will raise an error.

You shouldn't need to use this method during typical operations, but it may be useful in advanced applications.

Determining Whether A Session Data Object Exists For The Browser Id Associated With This Request

If you wish to determine whether a session data object with a key that is the current request's browser id exists in the session data manager's associated session data container, you can use the `hasSessionData()` method of the session data manager. This method returns true if there is session data associated with the current browser id:

```
<dtml-if "session_data_manager.hasSessionData(">
  The sessiondatamanager object has session data for the browser id
  associated with this request.
<dtml-else>
  The sessiondatamanager object does not have session data for
  the browser id associated with this request.
</dtml-if>
```

The `hasSessionData()` method is useful in highly customized applications, but is probably less useful otherwise. It is recommended that you use `REQUEST.SESSION` instead, allowing the session data manager to determine whether or not to create a new data object for the current request.

Embedding A Browser Id Into An HTML Link

You can embed the browser id name/value pair into an HTML link for use during HTTP GET requests. When a user clicks on a link with a URL encoded with the browser id, the browser id will be passed back to the server in the REQUEST.form namespace. If you wish to use formvar-based session tracking, you will need to encode all of your "public" HTML links this way. You can use the `encodeUrl()` method of browser id managers in order to perform this encoding:

```
<html>
<body>
<a href="<dtml-var "REQUEST.SESSION.getBrowserIdManager().encodeUrl('/amethod')">">Here</a>
  is a link.
</body>
</html>
```

The above dtml snippet will encode the URL `/amethod` (the target of the word "Here") with the browser id name/value pair appended as a query string. The rendered output of this DTML snippet would look something like this:

```
<html>
<body>
<a href="/amethod?_ZopeId=7HJhy78978979JHK">Here</a>
  is a link.
</body>
</html>
```

You may successfully pass URLs which already contain query strings to the `encodeUrl()` method. The `encodeUrl` method will preserve the existing query string and append its own name/value pair.

You may choose to encode the browser id into the URL using an "inline" style if you're checking for browser ids in the URL (e.g. if you've checked `URLs` in the "Look for Browser Id in" form element of your browser id manager):

```
<html>
<body>
<a href="<dtml-var "REQUEST.SESSION.getBrowserIdManager().encodeUrl('/amethod', style='inline')">">Here</a>
  is a link.
</body>
</html>
```

The above dtml snippet will encode the URL `/amethod` (the target of the word "Here") with the browser id name/value pair embedded as the first two elements of the URL itself. The rendered output of this DTML snippet would look something like this:

```
<html>
<body>
<a href="/_ZopeId/7HJhy78978979JHK/amethod">Here</a>
  is a link.
</body>
</html>
```

Using Session onAdd and onDelete Events

The configuration of a Transient Object Container (aka a session data container) allows a method to be called when a session data object is created (`onAdd`) or when it is invalidated or timed out (`onDelete`). The events are independent of each other. A session data manager can define, for example, an `onAdd` event but no `onDelete` event for the session data objects it manages, and vice versa. Or it can define both or neither events.

Why is this useful? It is advantageous to be able to prepopulate a session data object with "default" values before it's used by application code. You can use a session `onAdd` event to populate the session data object with default values. It's also sometimes advantageous to be able to write the contents of a session data object out to a permanent data store before it is timed out or invalidated. You can use a session `onDelete` event for this.

An `onAdd` or `onDelete` event for a session data object is defined by way of specifying the physical path to a callable Zope object in the "Script to call when objects are added" (`onAdd`), or "Script to call when objects are deleted"

(onDelete) in the *Manage* tab of the default transient object container at `/temp_folder/session_data` .

The "script to call" string is the Zope "physical path" of a specially-written External Method or Python Script which can perform an action on the contents of the data object at event time. For example, if you've written a method which aims to prepopulate a session data object named "onaddmethod" in the root of your Zope instance, you would set the onAdd method path on the Settings screen to `/onaddmethod`. Likewise, if you've written a method which does post-processing on the contents of a session data object named "ondeletemethod" in a folder of the Zope root named "afolder", you would set the onDelete method path in the Settings screen to `/afolder/ondeletemethod`. See the section below "Writing onAdd and onDelete Methods" for an introduction to writing onAdd and onDelete methods.

onAdd and onDelete events do not raise exceptions if logic in the method code fails. Instead, an error is logged in the Zope event log. You can see debug messages in the log if you've turned on debug logging via setting the "EVENT_LOG_FILE" environment variable to a filename as documented in the chapter entitled Installing and Starting Zope .

Writing onAdd and onDelete Methods

Session data objects optionally call a Zope method when they are created and when they are timed out or invalidated.

Specially-written Script (Python) scripts or External Methods can be written to serve the purpose of being called on session data object creation and invalidation.

The Script (Python) or External Method should define two arguments, "sdo" and "toc". "sdo" represents the session data object being created or terminated, and "toc" represents the transient object container in which this object is stored.

For example, to create a method to handle a session data object onAdd event which prepopulates the session data object with a DateTime object, you might write a Script (Python) named `onAdd` which had function parameters "sdo" and "toc" and a body of:

```
sdo['date'] = context.ZopeTime()
```

If you set the path to this method as the onAdd event, before any application handles the new session data object, it will be prepopulated with a key `date` that has the value of a DateTime object set to the current time.

To create a method to handle a session onDelete event which writes a log message, you might write an External Method with the following body:

```
from zLOG import LOG, WARNING
def onDelete(sdo, toc):
    logged_out = sdo.get('logged_out', None)
    if logged_out is None:
        LOG('session end', WARNING,
            'session ended without user logging out!')
```

If you set the path to this method as the onDelete event, a message will be logged if the `logged_out` key is not found in the session data object.

Note that for onDelete events, there is no guarantee that the onDelete event will be called in the context of the user who originated the session! Due to the "expire-after-so-many-minutes-of-inactivity" behavior of session data containers, a session data object onDelete event initiated by one user may be called while a completely different user is visiting the application. Your onDelete event method should not naively make any assumptions about user state. For example, the result of the Zope call `getSecurityManager().getUser()` in an onDelete session event method will almost surely *not* be the user who originated the session.

The session data object `onAdd` method will always be called in the context of the user who starts the session.

For both `onAdd` and `onDelete` events, it is almost always desirable to set proxy roles on event methods to replace the roles granted to the executing user when the method is called because the executing user will likely not be the user for whom the session data object was generated. For more information about proxy roles, see the chapter entitled *Users and Security*.

For additional information about using session `onDelete` events in combination with data object timeouts, see the section entitled "Session Data Object Expiration Considerations" in the *Concepts and Caveats* section below.

Configuration and Operation

Setting Initial Transient Object Container Parameters

Because the initial transient object container at `/temp_folder/session_data` is stored in a RAM database, it disappears and is recreated after each restart of your Zope server. This means that if you change one of its parameters, such as its timeout minutes setting, that change will be lost the next time you restart your Zope server.

To work around this problem, several environment variables may be specified during Zope startup which effect the parameters of the `session_data` transient object container that gets created in the `temp_folder`. These are:

`ZSESSION_ADD_NOTIFY`

An optional full Zope path name of a callable object to be set as the "script to call on object addition" of the `session_data` transient object container created in `temp_folder` at startup.

`ZSESSION_DEL_NOTIFY`

An optional full Zope path name of a callable object to be set as the "script to call on object deletion" of the `session_data` transient object container created in `temp_folder` at startup.

`ZSESSION_TIMEOUT_MINS`

The number of minutes to be used as the "data object timeout" of the `/temp_folder/session_data` transient object container.

`ZSESSION_OBJECT_LIMIT`

The number of items to use as a "maximum number of subobjects" value of the `/temp_folder` session data transient object container.

Instantiating Multiple Browser Id Managers (Optional)

Though you'll likely interact mostly with transient data objects while you develop session-aware code, these objects depend on a session data manager, which in turn depends on a browser id manager. A browser id manager is an object which doles out and otherwise manages browser ids. All session data managers need to talk to a browser id manager to get browser id information.

You needn't create a browser id manager to use sessioning. One is already created as a result of the initial Zope installation. If you've got special needs, you may want to instantiate more than one browser id manager. Having multiple browser id managers may be useful in cases where you have a "secure" section of a site and an "insecure" section of a site, each using a different browser id manager with respectively restrictive security settings. Some special considerations are required for this setup.

If you wish to add a different browser id manager anywhere in your Zope tree (for example, if you want to have two different virtual hosted sites that manage different browser ids), you may. However, once you've instantiated one browser id manager or if you keep the default browser id manager, you will not be able to instantiate another browser id manager in a place where the new browser id manager can acquire the original browser id manager via its containment path (for programmers: the session id manager's class' Zope `__replaceable__` property is set to `UNIQUE`). This means, practically, that if you wish to have multiple browser id managers, you need to carefully think about where they should go, and then you'll need to *delete* the default root browser id manager, place new ones in the most deeply-nested containers first, working your way out towards the root.

In the container of your choosing, select "Browser Id Manager" from the add dropdown list in the Zope management interface. When you add a new browser id manager, the form options available are:

Id — you cannot choose an `id` for your browser id manager. It must always be "browser_id_manager". Additionally, you cannot rename a browser id manager. This is required in the current implementation so that session data managers can find session id managers via Zope acquisition.

Title — the browser id manager title.

Browser Id Name — the name used to look up the value of the browser id. This will be the name looked up in the `cookies` or `form` REQUEST namespaces when the browser id manager attempts to find a cookie, form variable, or URL with a browser id in it.

Look for Browser Id Name In — choose the request elements to look in when searching for the browser id name. You may choose "cookies", "Forms and Query Strings", and "URLs".

Automatically Encode Zope-Generated URLs With A Browser Id — if this option is checked, all URLs generated by Zope (such as URLs obtained via the `absolute_url` method of all Zope objects) will have a browser id name/value pair embedded within them. This typically only make sense if you've also got the `URLs` setting of "Look for Browser Id in" checked off.

Cookie Path — this is the `path` element which should be sent in the browser id cookie. For more information, see the Netscape Cookie specification at http://home.netscape.com/newsref/std/cookie_spec.html.

Cookie Domain — this is the "domain" element which should be sent in the browser id cookie. For more information, see the Netscape Cookie specification at http://home.netscape.com/newsref/std/cookie_spec.html. Leaving this form element blank results in no domain element in the cookie. If you change the cookie domain here, the value you enter must have at least two dots (as per the cookie spec).

Cookie Lifetime In Days — browser id cookies sent to browsers will last this many days on a remote system before expiring if this value is set. If this value is 0, cookies will persist on client browsers for only as long as the browser is open.

Only Send Cookie Over HTTPS — if this flag is set, only send cookies to remote browsers if they're communicating with us over https. The browser id cookie sent under this circumstance will also have the `secure` flag set in it, which the remote browser should interpret as a request to refrain from sending the cookie back to the server over an insecure (non-https) connection. NOTE: In the case you wish to share browser id cookies between https and non-https connections from the same browser, do not set this flag.

After reviewing and changing these options, click the "Add" button to instantiate a browser id manager.

You can change any of a browser id manager's initial settings by visiting it in the management interface.

Instantiating A Session Data Manager (Optional)

After instantiating at least one browser id manager, it's possible to instantiate a session data manager. You don't need to do this in order to begin using Zope's sessioning machinery, as a default session data manager is created as `/session_data_manager` . Creating a separate session data manager in a different location in case you want to customize sessioning-related behavior in that place is possible, however.

You can place a session data manager in any Zope container, as long as a browser id manager object named `browser_id_manager` can be acquired from that container. The session data manager will use the first acquired browser id manager.

Choose "Session Data Manager" within the container you wish to house the session data manager from the "Add" dropdown box in the Zope management interface.

The session data manager add form displays these options:

Id — choose an id for the session data manager

Title — choose a title for the session data manager

Transient Object Container Path — enter the Zope path to a Transient Object Container in this text box in order to use it to store your session data objects. An example of a path to a Zope transient object container is `/temp_folder/session_data` .

After reviewing and changing these options, click the "Add" button to instantiate a session data manager.

You can manage a session data manager by visiting it in the management interface. You may change all options available during the add process by doing this.

Instantiating a Transient Object Container

The default transient object container at `/temp_folder/session_data` stores its objects in RAM, so these objects disappear when you restart Zope. If you want your session data objects to persist across server reboots, or if you have a potentially very large collection of session data objects, or if you'd like to share sessions between ZEO clients, you will want to instantiate a transient data container in a more permanent storage. A heavily-utilized transient object container *should be instantiated inside a database which is nonundoing* ! Although you may instantiate a transient data container in any storage, if you make heavy use of an external session data container in an undoing database (such as the default Zope database which is backed by "FileStorage", an undoing and versioning storage), your database will grow in size very quickly due to the high-write nature of session tracking, forcing you to pack very often.

For a product which allows you to use a mounted nonundoing database, see Shane Hathaway's ExternalMount product .

Here are descriptions of the add form of a Transient Object Container, which may be added by selecting "Transient Object Container" for the Zope Add list.:

Id — the id of the transient object container

Title (optional) — the title of the transient object container

Data object timeout in minutes — enter the number of minutes of inactivity which causes a contained transient object be be timed out. "0" means no expiration.

Maximum number of subobjects — enter the maximum number of transient objects that can be added to this transient object container. This value helps prevent "denial of service" attacks to your Zope site by effectively limiting the number of concurrent sessions.

Script to call upon object add (optional) — when a session starts, you may call an external method or Script (Python). This is the Zope path to the external method or Script (Python) object to be called. If you leave this option blank, no onAdd function will be called. An example of a method path is `/afolder/amethod` .

Script to call upon object delete (optional) — when a session ends, you may call an external method or Script (Python). This is the Zope path to the external method or Script (Python) object to be called. If you leave this option blank, no onDelete function will be called. An example of a method path is `/afolder/amethod` .

Multiple session data managers can make use of a single transient object container to the extent that they may share the session data objects placed in the container between them. This is not a recommended practice, however, as it has not been tested at all.

The `data object timeout in minutes` value is the number of minutes that session data objects are to be kept since their last-accessed time before they are flushed from the data container. For instance, if a session data object is accessed at 1:00 pm, and if the timeout is set to 20 minutes, if the session data object is not accessed again by 1:19:59, it will be flushed from the data container at 1:20:00 or a time shortly thereafter. "Accessed", in this terminology, means "pulled out of the container" by a call to the session data manager's `getSessionData()` method or an equivalent (e.g. a reference to `REQUEST.SESSION`). See "Session Data Object Expiration Considerations" in the Concepts and Caveats section below for details on session data expiration.

Randall Kern has additionally written a ZEO + sessioning How-To that may help, although it describes an older generation of Zope sessioning machinery, so you may need to extrapolate a bit.

Configuring Sessioning Permissions

You need only configure sessioning permissions if your requirements deviate substantially from the norm. In this case, here is a description of the permissions related to sessioning:

Permissions related to browser id managers:

Add Browser Id Manager — allows a role to add browser id managers. By default, enabled for `Manager` .

Change Browser Id Manager — allows a role to change an instance of a browser id manager. By default, enabled for `Manager` .

Access contents information — allows a role to obtain data about browser ids. By default, enabled for `Manager` and `Anonymous` .

Permissions related to session data managers:

Add Session Data Manager — allows a role to add session data managers. By default, enabled for `Manager` .

Change Session Data Manager — allows a role to call management-related methods of a session data manager. By default, enabled for `Manager` .

Access session data — allows a role to obtain access to the session data object related to the current browser id. By default, enabled for `Manager` and `Anonymous` . You may wish to deny this permission to roles who have DTML or Web-based Python scripting capabilities who should not be able to access session data.

Access arbitrary user session data — allows a role to obtain and otherwise manipulate any session data object for which the browser id is known. By default, enabled for `Manager` . (For more information, see the `getSessionDataByKey` method described in the sessioning API in the Zope Help System.)

Access contents information — allows a role to obtain data about session data. By default, enabled for `Manager` and `Anonymous` .

Permissions related to transient object containers:

Add Transient Object Container — allows a role to add transient objects containers. By default, enabled for `Manager` .

Change Transient Object Container — allows a role to make changes to a transient object container.

Access Transient Objects — allows a role to obtain and otherwise manipulate the transient object related to the current browser id.

Concepts and Caveats

Security Considerations

Sessions are insecure by their very nature. If an attacker gets a hold of someone's browser id, and if they can construct a cookie or use form elements or URL elements to pose as that user from their own browser, they will have access to all information in that user's session. Sessions are not a replacement for authentication for this reason.

Ideally, you'd like to make certain that nobody but the user its intended for gets a hold of his browser id. To take steps in this direction, and if you're truly concerned about security, you will ensure that you use cookies to maintain browser id information, and you will secure the link between your users and your site using SSL. In this configuration, it is more difficult to "steal" browser id information as the browser id will not be evident in the URL and it will be very difficult for attackers to "tap" the encrypted link between the browser and the Zope site.

There are significant additional risks to user privacy in employing sessions in your application, especially if you use URL-based or formvar-based browser ids. Commonly, a browser id is embedded into a form/querystring or a URL in order to service users who don't have cookies turned on.

For example, this kind of bug was present until recently in a lot of webmail applications: if you sent a mail to someone that included a link to a site whose logs you could read, and the user clicked on the link in his webmail page, the full URL of the page, including the authentication (stored as session information in the URL) would be sent as a HTTP REFERER to your site.

Nowadays all serious webmail applications either choose to store at least some of the authentication information outside of the URL (in a cookie for instance), or process all the user-originated URLs included in the mail to make them go through a redirection that sanitizes the HTTP REFERER.

The moral of the story is: if you're going to use sessions to store sensitive information, and you link to external sites within your own site, you're best off using *only* cookie-based browser ids.

Browser Id (Non-)Expiration

Browser ids do not actually themselves expire. They persist for as long as their conveyance mechanism allows. For example, a browser id will last for as long as the browser id cookie persists on the client, or for as long as someone

uses a bookmarked URL with a browser id encoded into it. The same id will be obtained by a browser id manager on every visit by that client to a site - potentially indefinitely depending on which conveyance mechanisms you use and your configuration for cookie persistence.

In lieu of expiry of browser ids, the transient object container which holds session data objects implements a policy for data object expiration. If asked for a session data object related to a particular browser id which has been expired by a session data container, a session data manager will return a new session data object.

Session Data Object Expiration Considerations

Session data objects expire after the period between their last access and "now" exceeds the timeout value provided to the session data container which hold them. No special action need be taken to expire session data objects.

However, because Zope has no scheduling facility, the sessioning machinery depends on the continual exercising of itself to expire session data objects. If the sessioning machinery is not exercised continually, it's possible that session data objects will stick around longer than the time specified by their data container timeout value. For example:

- User A exercises application machinery that generates a session data object. It is inserted into a session data container which advertises a 20-minute timeout.
- User A "leaves" the site.
- 40 minutes go by with no visitors to the site.
- User B visits 60 minutes after User A first generated his session data object, and exercises app code which hands out session data objects. *User A's session is expired at this point, 40 minutes "late".*

As shown, the time between a session's onAdd and onDelete is not by any means *guaranteed* to be anywhere close to the amount of time represented by the timeout value of its session data container. The timeout value of the data container should only be considered a "target" value.

Additionally, even when continually exercised, the sessioning machinery has a built in error potential of roughly 20% with respect to expiration of session data objects to reduce resource requirements. This means, for example, if a transient object container timeout is set to 20 minutes, data objects added to it may expire anywhere between 16 and 24 minutes after they are last accessed.

Sessioning and Transactions

Sessions interact with Zope's transaction system. If a transaction is aborted, the changes made to session data objects during the transaction will be rolled back.

Mutable Data Stored Within Session Data Objects

If you mutate an object stored as a value within a session data object, you'll need to notify the sessioning machinery that the object has changed by calling `set` or `__setitem__` on the session data object with the new object value. For example:

```
session = self.REQUEST.SESSION
foo = {}
foo['before'] = 1
session.set('foo', foo)

# mutate the dictionary
foo['after'] = 1
```

```
# performing session.get('foo') 10 minutes from now will likely
# return a dict with only 'before' within!
```

You'll need to treat mutable objects immutably, instead. Here's an example that makes the intent of the last example work by doing so:

```
session = self.REQUEST.SESSION
foo = {}
foo['before'] = 1
session.set('foo', foo)

# mutate the dictionary
foo['after'] = 1

# tickle the persistence machinery
session.set('foo', foo)
```

An easy-to-remember rule for manipulating data objects in session storage: always explicitly place an object back into session storage whenever you change it. For further reference, see the "Persistent Components" chapter of the Zope Developer's Guide at <http://www.zope.org/Documentation/ZDG>.

Session Data Object Keys

A session data object has essentially the same restrictions as a Python dictionary. Keys within a session data object must be hashable (strings, tuples, and other immutable basic Python types; or instances which have a `__hash__` method). This is a requirement of all Python objects that are to be used as keys to a dictionary. For more information, see the associated Python documentation at <http://www.python.org/doc/current/ref/types.html> (Mappings -> Dictionaries).

In-Memory Session Data Container RAM Utilization

Each session data object which is added to an "internal" (RAM-based) session data container will consume at least 2K of RAM.

Mounted Transient Object Container Caveats

Persistent objects which have references to other persistent objects in the same database cannot be committed into a mounted database because the ZODB does not currently handle cross-database references.

Transient object containers which are sometimes stored in a "mounted" database (as is currently the case for the default `/temp_folder/session_data` TOC. If you use a transient object container that is accessed via a "mounted" database, you cannot store persistent object instances which have already been stored in the "main" database as keys or values in a session data object. If you try to do so, it is likely that an `InvalidObjectReference` exception will be raised by the ZODB when the transaction involving the object attempts to commit. As a result, the transaction will fail and the session data object (and other objects touched in the same transaction) will fail to be committed to storage.

If your "main" ZODB database is backed by a nonundoing storage, you can avoid this condition by storing session data objects in an transient object container instantiated within the "main" ZODB database. If this is not an option, you should ensure that objects you store as values or keys in a session data object held in a mounted session data container are instantiated "from scratch" (via their constructors), as opposed to being "pulled out" of the main ZODB.

Conflict Errors

This session tracking software stores all session state in Zope's ZODB. The ZODB uses an optimistic concurrency strategy to maintain transactional integrity for simultaneous writes. This means that if two objects in the ZODB are changed at the same time by two different connections (site visitors) that a "ConflictError" will be raised. Zope retries requests that raise a ConflictError at most 3 times. If your site is extremely busy, you may notice ConflictErrors in the Zope debug log (or they may be printed to the console from which you run Zope). An example of one of these errors is as follows:

```
2001-01-16T04:26:58 INFO(0) Z2 CONFLICT Competing writes at, /getData
Traceback (innermost last):
File /zope/lib/python/ZPublisher/Publish.py, line 175, in publish
File /zope/lib/python/Zope/__init__.py, line 235, in commit
File /zope/lib/python/ZODB/Transaction.py, line 251, in commit
File /zope/lib/python/ZODB/Connection.py, line 268, in commit
ConflictError: '\000\000\000\000\000\000\002/'
```

Errors like this in your debug log (or console if you've not redirected debug logging to a file) are normal to an extent. If your site is undergoing heavy load, you can expect to see a ConflictError perhaps every 20 to 30 seconds. The requests which experience conflict errors will be retried automatically by Zope, and the end user should *never* see one. Generally, session data objects attempt to provide application-level conflict resolution to reduce the limitations imposed by conflict errors NOTE: to take advantage of this feature, you must store your transient object container in a storage such as FileStorage or TemporaryStorage which supports application-level conflict resolution.

Zope Versions and Sessioning

In the default Zope sessioning configuration, session data objects are not versioned. This means that when you change a session data object while using a Zope Version, the changes will be "seen" outside of the version.

Further Documentation

All of the methods implemented by Session Data Managers, Browser Id Managers, Transient Data Containers and Transient Data objects are fully documented in the Zope help system under Zope Help -> API Reference -> Session API and Zope Help -> API Reference -> Transient Object.

Scalability and ZEO

When a web application receives more requests than it can handle over a short period of time, it can become unresponsive. In the worst case, too many concurrent requests to a web application can cause the software which services the application to crash. This can be a problem for any kind of web-based app, not just those which are served by Zope.

The obvious solution to this problem is to use more than one server. When one server becomes overloaded, the others can then hopefully continue to successfully serve requests. By adding additional servers to this kind of configuration, you can "scale" your web application as necessary to meet demand.

Using multiple servers has obvious benefits, but it also poses serious challenges. For example, if you have five servers, then you must ensure that all five server installations are populated with the same information. This is not a very hard task if you have only a few static web pages, but for larger applications with large bodies of rapidly changing information, manually synchronizing the data which drives five separate server installations is almost impossible, even with the "out of the box" features that Zope provides.

A "stock" Zope installation uses the Zope Object Database as its content store, using a "storage" which is named a "FileStorage". This storage type (there are others) keeps all of your Zope data in a single file on your computer's hard drive, typically named `Data.fs`. This configuration works well until you need to add an additional Zope server to your site to handle increased traffic to your web application. Two Zope servers cannot share this file. The file is "locked" by one Zope server and no other Zope server can access the file. Thus, in a "stock" Zope configuration, it is impossible to add Zope servers which read from the same database in order to "scale" your web application to meet demand.

To solve this problem, Zope Corporation has created another kind of "storage", which operates using a client/server architecture, allowing many Zopes to share the same database information. This product is known as 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 share a Zope Object Database between more than one Zope process. By using ZEO, you may run multiple instances of Zope on a single computer or on multiple computers. Thus, you may spread requests to your web application between Zope servers. You may add more computers as the number of requests grows, allowing your web application to scale. Furthermore, if one Zope server fails or crashes, other servers can still service requests while you fix the broken one. ZEO takes care of making sure each Zope installation uses consistent information from the same Zope Object Database.

ZEO uses a client/server architecture. The Zope processes (shown on multiple computers in the diagram below) are the *ZEO Clients*. All of the clients connect to one, central *ZEO Storage Server*, as shown in the image below.

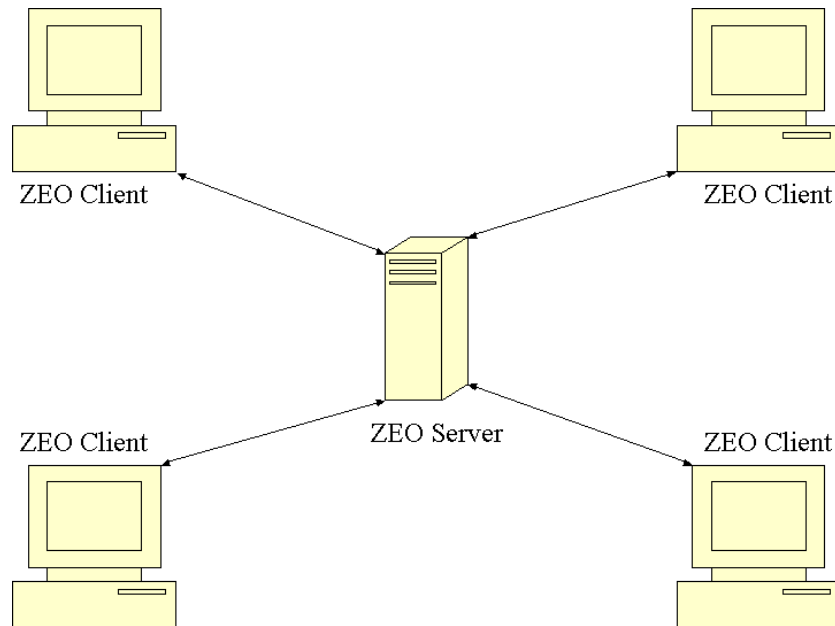


Figure 20-1 Simple ZEO illustration

The terminology may be a bit confusing. Typically, you may think of Zope as a server, not a client. But 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 to disparate geographic locations. In this chapter we'll explore some interesting ways you can distribute your ZEO clients.

When you should use ZEO

A ZEO-configured Zope installation has the capability to serve large numbers of requests in short periods of time. If your site is only moderately trafficked and you get no complaints about the responsiveness of the site, you probably don't need ZEO. You may need ZEO if:

- You've got a single Zope server in production which cannot service application demand within a reasonable amount of time. Zope is a high-performance system, and one Zope can handle millions of hits per day, but there are upper bounds on the capacity of a single Zope server. ZEO allows you to scale your site by adding more hardware on which you may place extra Zope servers to handle excess demand.
- Your site is critical and requires 24/7 uptime. Using ZEO can help you add redundancy to your server configuration.
- You want to distribute your site to disparate geographic locations in order to increase response time to remote sites. ZEO allows you to place Zope servers which use the same ZODB in separate geographic locations.
- You want to "debug" an application which is currently served by a single Zope server from another Zope process. This is an advanced technique useful to Python developers, but is not covered in this book.

Installing, configuring, and maintaining a ZEO-enabled Zope requires some system administration knowledge. 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

ZEO is not distributed with Zope. You may download it from the Products Section of Zope.org. The most recent version of ZEO as of this writing is ZEO 2.0.

There are some prerequisites before you will be successfully able to use ZEO:

- All of the Zope servers in a ZEO-enabled configuration must run the same version of Zope and ZEO. A ZEO 1.0 client will not work against a ZEO 2.0 server and vice versa. Likewise, older versions of Zope may not work with the latest version of ZEO. The easiest way to meet this prerequisite is to make sure all of your computers use the latest versions of Zope and ZEO.
- 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.

Installing ZEO requires some manual preparation. To install ZEO, download the latest ZEO package (as of this writing it is at <http://www.zope.org/Products/ZEO/ZEO-2.0.tar.gz/view>), 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-X.X.tar.gz
```

On Unix that does not have GNU tar, use the following command:

```
$ gzip -cd ZEO-X.X.tar.gz | tar -xvf -
```

On Windows, you can unpack the archive with WinZip.

Now you should have a *ZEO-X.X* directory on your hard disk. The next few steps install ZEO itself. Before installing ZEO, make sure you back up your Zope system first.

To make use of ZEO, you must install ZEO files into your Zope's top-level directory. This can be done on UNIX or Windows by issuing the following commands (under a shell on UNIX or in a DOS box on Windows):

```
$ cd ZEO-X.X
$ python2.1 setup.py install --home=/path/to/your/Zope/top/level/dir
```

Make sure you use the same Python interpreter which you're using to run Zope to invoke the `setup.py` command. When you're done with this step, you should have a "ZEO" directory inside your Zope's `lib/python` directory.

Next, you must create a special file in your "top-level" Zope directory named *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 hostname or IP address of the ZEO 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` and the port is `7700`.

You should now have ZEO properly installed. Try it out by first starting the server. Go to your Zope top level directory in a terminal window or DOS box and type:

```
python2.1 lib/python/ZEO/start.py -p 7700
```

This will start the ZEO server listening on TCP 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 Management*. 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 a single ZEO client 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 multiple ZEO clients, which is explained in the next section.

How to Run Multiple ZEO Clients

We can expand the capacity of the site by adding additional 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. Let's assume all of these computers exist within the ".zopezoo.org" Internet domain.

The first step is to run the ZEO server on *zooserver*. To tell your ZEO server to listen on the tcp socket at port `7700` on the *zooserver* interface, run the server with the *start.py* script like this:

```
$ python2.1 lib/python/ZEO/start.py -p 7700
```

This will start the ZEO server. Now, you can start up your clients by going to each client and configuring each of them by providing them with the following *custom_zodb.py*:

```
import ZEO.ClientStorage
Storage=ZEO.ClientStorage.ClientStorage(('zooserver.zopezoo.org',7700))
```

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 work. 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". On each line of this property, put the URL to your various ZEO clients, as shown in the figure below.

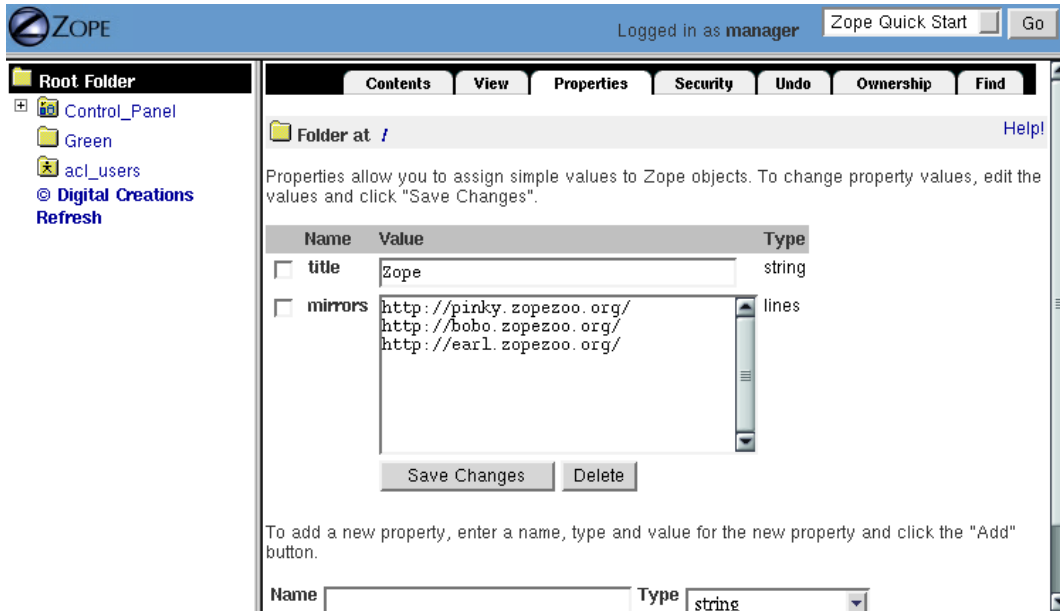


Figure 20-2 Figure of property with URLs to mirrors

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 mirrors>
    <li><a href="&dtml-sequence-item;"><dtml-var
sequence-item></a></li>
  </dtml-in>
</ul>
```

Or, in Script (Python):

```
## Script (Python) "generate_mirror"
##bind container=container
##bind context=context
##bind namespace=
##bind script=script
##bind subpath=traverse_subpath
##parameters=a, b
##title=
##
print "<h2>Please choose from the following mirrors: <ul>"
for mirror in container.mirrors:
    print "<li><a href=\"%s\">%s</a>" % (mirror, mirror)
return printed
```

This DTML (and Script (Python) equivalent) 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.

Alternately, 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 the figure below.

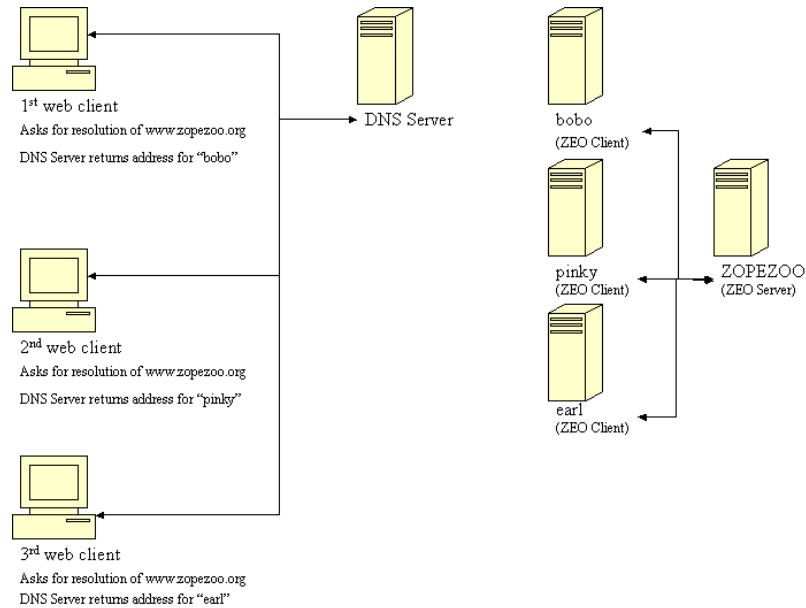


Figure 20-3 Load balancing with round-robin DNS.

When *www.zopezoo.org* gets resolved, DNS 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 information gets cached by the other nameservers on the Internet. 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 acceptable, because the total sum of the requests are really spread over your various ZEO clients.

One potential problem with this solution is that it can take hours or 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 an 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 an advanced technique that is largely 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 the figure below.

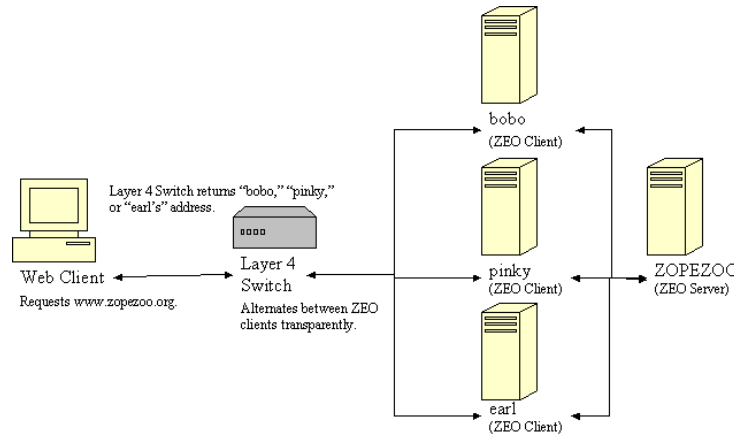


Figure 20-4 Illustration of Layer 4 switching

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 the Storage Server as A Single Point of Failure

Without ZEO, a single 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.

However, in a typical ZEO setup there is still a single point of failure: the ZEO server itself. Without using commercial software, this single point of failure cannot be removed.

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 one-hundred-percent uptime. 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 may help you to create a backup storage server. 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.

ZEO also has a commercial "multiple-server" configuration which provides for redundancy at the storage level. Zope Corporation sells a commercial product named Zope Replication Services that provides redundancy in storage server services. It allows a "secondary" storage server to take over for a "primary" server when the primary fails.

ZEO Server Details

The final piece of the puzzle is where the ZEO server 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?

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 the figure below.

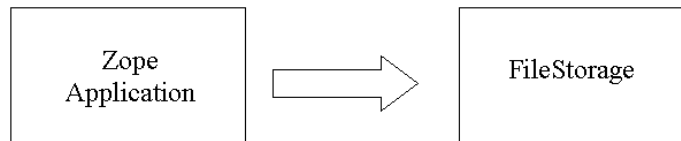


Figure 20-5 Zope connected to a filestorage

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 the figure below.

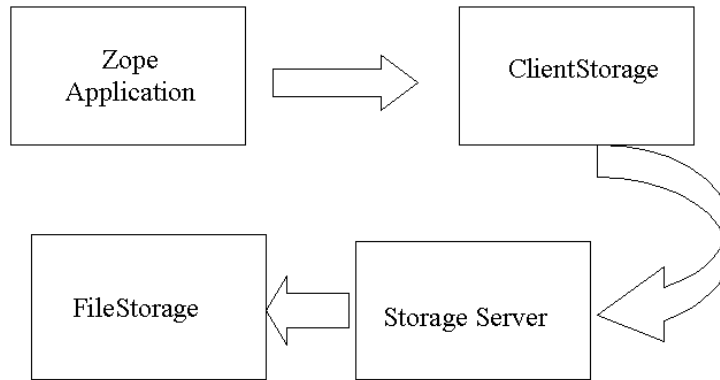


Figure 20-6 Zope with a Client Storage and Storage server

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. In a "stock" ZEO setup, this storage file is in the same place as it would be were you not running ZEO (within your Zope directory's `var` directory named `Data.fs`).

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 loses 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 *many* 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*. However, 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.

This is not an unsolvable 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.

Managing Zope Objects Using External Tools

So far, you've been working with Zope objects in your web browser via the Zope Management Interface. This chapter details how to use common non-browser-based common to access and modify your Zope content.

Editing Zope content and code in the Zope Management Interface is sometimes painful, especially when dealing with Python code, DTML, ZPT, or even just HTML. The standard TEXTAREA text manipulation widget provided by most browsers has an extremely limited feature set: no syntax highlighting, no autoindent, no key rebindings, no WYSIWYG HTML editing, and sometimes not even a search and replace function!

In short, people want to use their own tools, or at least more featureful tools, to work with Zope content.

It is possible under most operating systems to use the text "cut and paste" facility (Ctrl-C, Ctrl-V under Windows, for example) to move text between traditional text/HTML editors and your browser, copying data back and forth between the Zope Management interface and your other tools. This is, at best, cumbersome.

Luckily, Zope provides features that may allow you to interface Zope directly with your existing tools. This chapter describes these features, as well as the caveats for working with them.

General Caveats

Most external tools expect to deal with "file-like" content. Zope objects are not really files in the strict sense of the word so there are caveats to using external tools with Zope:

- Zope data is not stored in files in the filesystem. Thus, tools which only work on files will not work with Zope without providing a "bridge" between the tool and Zope's file-like representation of its object database. This "bridge" is typically accomplished using Zope's FTP or WebDAV features.
- Zope doesn't enforce any file extension rules when creating objects. Some tools don't deal well with objects that don't have file extensions in their names (notably Macromedia Dreamweaver). To avoid this issue, you may name your objects with file extensions according to their type (e.g. name all of your ZPT objects with an `.html` file extension), or use a tool that understands extensionless "files". However, this approach has numerous drawbacks.
- Creating new objects can sometimes be problematic. Because Zope doesn't have a default object-type-to-file-extension policy, new content will often be created as the wrong "kind" of object. For example, if you upload an HTML file "foo.html" via FTP to a place where "foo.html" did not previously exist, it will be created (by default) as a DTML Document object, whereas you may want it to be created as a Zope Page Template. Zope provides a facility to specify the object type created on a per-folder and per-request basis (PUT_factory) that is detailed in this chapter.
- External tools don't know about Zope object properties. If you modify an object in an external tool, it may forget its property list.
- Some external tools have semantics that can drive Zope crazy. For instance, some like to create backup files with an id that is invalid for Zope. Also, some tools will do a move-then-copy when saving, which creates a new Zope object that is divorced from the history of the original object.
- There is nowhere to send meaningful error messages. These integration features expect a finite set of errors defined by the protocol. Thus, the actual problem reported by Zope, such as a syntax error in a page template,

cannot be displayed to the user.

- The interactions between the tools and Zope can vary widely. On the client side, different versions of software have different bugs and features. For instance, using FTP under Emacs will sometimes work by default, but sometimes it needs to be configured. Also, Microsoft has many different implementations of DAV in Windows and Office, each with changes that make life difficult. Finally, Zope itself has substantially improved support between Zope 2.3 and Zope 2.6.
- Finally, the semantics of Zope can interfere with the experience. The same file on your harddrive, when copied into www.zope.org and your local copy of Zope, will have different results. In the case of the CMF, Zope will actually alter what you saved (to add metadata).

These caveats aside, you may use traditional file manipulation tools to manage most kinds of Zope objects.

FTP and WebDAV

Most Zope "file-like" objects like DTML Methods, DTML Documents, Zope Page Templates, Script (Python) objects and others can be edited with FTP and WebDAV. 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 files from one computer to another. Many text editors and HTML editors support FTP.

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 and HTML editors as FTP.

Greg Stein's excellent webdav.org site has an FAQ that introduces WebDAV. The FAQ provides a comparison of DAV to FTP.

Using FTP to Manage Zope Content

There are many popular FTP clients, and many web browsers like Netscape and Microsoft Internet Explorer come with FTP clients. Many text and HTML editors also directly support FTP. You can make use of these clients to manipulate Zope objects via FTP.

Determining Your Zope's FTP Port

In the chapter entitled "Using the Zope Management Interface", you determined the HTTP port of your Zope system by looking at Zope's start-up output. You can find your Zope's FTP port by following the same 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*. If Zope doesn't report an "FTP server started", it likely means that you need to turn Zope's FTP server on by using the `-f` command line option to the `start` script as detailed in the chapter entitled *Installing and Starting Zope*.

Transferring Files with WS_FTP

WS_FTP is a popular FTP client for Windows that you can use to transfer documents and files between Zope and your local computer. WS_FTP can be downloaded from the Ipswitch Home Page.

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 the figure below.

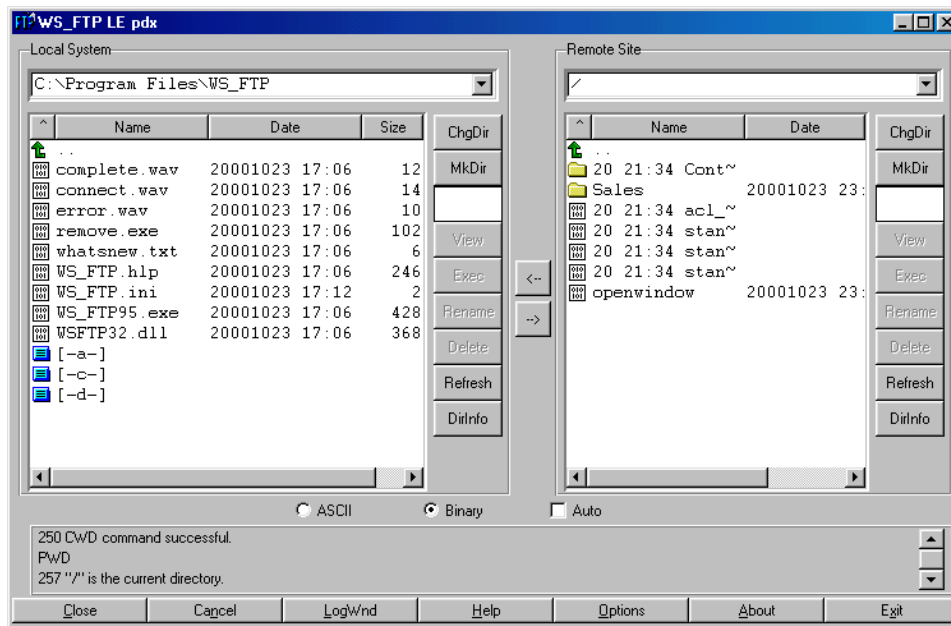


Figure 5-1 Viewing the Zope object hierarchy through FTP

Transferring files to and from Zope is straightforward when using 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 a matter of selecting the file you want to transfer and clicking either the left arrow (download) or the right arrow (upload).

You may transfer Zope objects to your local computer as files using WS_FTP. You may then edit them and upload them to Zope again when you're finished.

Remote Editing with FTP/DAV-Aware Editors

Editing Zope Objects with Emacs FTP Modes

Emacs is a very popular text editor. Emacs comes in two major "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 a fairly straightforward matter. More importantly, you need not leave Emacs in order to edit content that lives inside your Zope.

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 directory in XEmacs, press Ctrl-X D and enter a directory specification in the form:

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

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

Emacs — To visit a remote directory in Emacs, press Ctrl-X D and enter a directory specification in 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 before displaying the directory contents. The directory contents of the root folder will look a little like the picture below:

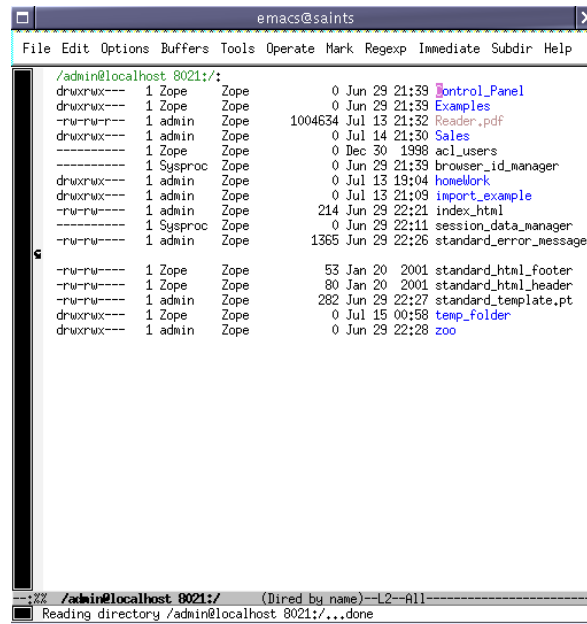


Figure 5-2 Viewing the Zope Root Folder via ange-ftp

You can visit any of these "files" (which are really Zope objects) by selecting them in the usual Emacs way: enter to select, modify the file, Ctrl-X S to save, etc. You can even create new "files" by visiting a file via "Ctrl-X Ctrl-F". New files will be created as DTML Document objects unless you have a PUT_factory (described below) installed to specify a different kind of initial object.

The ftp program that ships with Microsoft Windows is incompatible with NTEmacs (the Windows NT version of GNU Emacs). To edit Zope objects via "ange-ftp" under NTEmacs, it requires that you have a special FTP program. This program ships with "Cygwin", a UNIX implementation for Windows. To use NTEmacs download and install Cygwin and add the following to your .emacs configuration file:

```
(setq ange-ftp-ftp-program-name "/cygwin/bin/ftp.exe")
(setq ange-ftp-try-passive-mode t)
(setq ange-ftp-ftp-program-args '("-i" "-n" "-g" "-v" "--prompt" ""))
```

For another perspective on using Emacs with Zope, see the Zope.org HowTo written by Miklos Nemeth . There's even a DTML mode for XEmacs and a ZWiki mode for Xemacs , both by "albut".

Caveats With FTP

In addition to the general caveats listed above, using FTP with Zope has some unique caveats:

- You need to be aware of passive mode for connecting to Zope.
- The "move-then-copy" problem is most apparent when using Emacs' ange-ftp.

Editing Zope Objects with WebDAV

WebDAV is an extension to the HTTP protocol that provides features that allow users to concurrently author and edit content on web sites. WebDAV offers features like locking, revision control, and the tagging of 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 a fairly long time.

WebDAV is a newer Internet protocol compared to HTTP or FTP, so there are fewer clients that support it. There is, however, growing momentum behind the WebDAV movement and more clients are being developed rapidly.

The WebDAV protocol is evolving quickly, and new features are being added all the time. You can use any WebDAV client to edit your Zope objects by simply pointing the client at your object'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 DTML or ZPT objects, 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 to the Zope `start` script. 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.

For more information about starting Zope with a WebDAV source port turned on, see the chapter entitled Installing and Starting Zope . The "standard" WebDAV source port number (according to IANA) is 9800.

Unfortunately, this entire discussion of source vs. rendered requests is too esoteric most users, who will try the regular port. Instead of breaking, it will work in very unexpected ways, leading to confusion. Until DAV clients support the standard's provision for discovering the source URL, this distinction will have to confronted.

Note

Zope 2.6 has optional support for returning the source version of a resource on the normal HTTP port. It does this by inspecting the user agent header of the HTTP request. If the user agent matches a string you have configured into your server settings, the source is returned.

This is quite useful, as there are few cases in which authoring tools such as cadaver or Dreamweaver will want the rendered version. For more information on this optional support, read the section "Environment Variables That Affect Zope At Runtime" in Installing and Starting Zope .

One program that supports WebDAV is a command-line tool named `cadaver` . It is available for most UNIX systems (and Cygwin under Windows) from WebDAV.org .

It is typically invoked from a command-line using the command `cadaver` against Zope's WebDAV "source port":

```
$ cadaver
dav:!!> open http://saints.homeunix.com:9800/
Looking up hostname... Connecting to server... connected.
Connecting to server... connected.
dav:/> ls
Listing collection `/' : (reconnecting...done) succeeded.
Coll: Control_Panel          0 Jun 14:03
Coll: Examples               0 Jun 14:01
Coll: ZopeBook               0 Jul 22:57
Coll: temp                   0 Jul 2002
Coll: temp_folder            0 Jul 19:47
Coll: tutorial                0 Jun 00:42
Coll: acl_users              0 Dec 1998
Coll: browser_id_manager     0 Jun 14:01
Coll: index_html             93 Jul 01:01
Coll: session_data_manager   0 Jun 14:01
Coll: standard_error_message 1365 Jan 2001
Coll: standard_html_footer   53 Jan 2001
Coll: standard_html_header   80 Jan 2001
Coll: standard_template.pt   282 Jun 14:02
dav:/>
```

Cadaver allows you to invoke an editor against files while inside the command-line facility:

```
dav:/> edit index_html
Connecting to server... connected.
Locking `index_html': Authentication required for Zope on server `saints.homeunix.com':
Username: admin
Password:
Retrying: succeeded.
Downloading `/index_html' to /tmp/cadaver-edit-001320
Progress: [=====>] 100.0% of 93 bytes succeeded.
Running editor: `vi /tmp/cadaver-edit-001320'...
```

In this case, the `index_html` object was pulled up for editing inside of the `vi` text editor. You can specify your editor of choice on most UNIX-like systems by changing the `EDITOR` environment variable.

You can also use `cadaver` to transfer files between your local directory and remote Zope, as described above for `WS_FTP`. For more advanced synchronization of data, the `sitcopy` program can inspect your local and remote data and only transfer the changes, using FTP or DAV.

Other commercial applications, such as Macromedia Dreamweaver and Microsoft Office also support WebDAV. For more information regarding programs which support the WebDAV protocol, see WebDAV.org .

Using a PUT_factory to Specify the Type of Objects Created With FTP and DAV

Because Zope is an "file-extensionless" system, it is often necessary to tell Zope to create a specific kind of object when an FTP or WebDAV client causes a new object to be entered into the Zope system. This action is typically called a "PUT" (the name of the FTP and DAV command that causes a file to be uploaded).

The default policy for new Zope object creation is as follows:

If the content type is	Create this kind of object
----- 'text/{anything}'	----- create a DTML Document
'image/{anything}'	create an Image object
'{anything else}'	create a File object

Zope allows you to override its default behavior by creating a Python Script or External Method named "PUT_factory" in the folder in and under which you want the new behavior to take effect.

The following example is a bit of text suitable as an External Method that you can put in your "root" folder which causes Zope to create Zope Page Templates instead of DTML Documents when it encounters a PUT of type "text/html" or "text/plain" (or any other kind of textual content):

```
from Products.PageTemplates.ZopePageTemplate import ZopePageTemplate

def PUT_factory(self, name, typ, body):
    if typ.startswith('text'):
        return ZopePageTemplate( name, text=body, content_type=typ )
```

Put this text into a file on your Zope server's filesystem in your Zope installation's `Extensions` folder named `PUT_factory.py`. Then create an External Method object in your root folder with an id of `PUT_factory`, a title of "PUT factory for Page Templates" a Module Name of `PUT_factory` and a Function Name of `PUT_factory` as well. Once this External Method has been created in the root folder, any new file that is uploaded via FTP or DAV with a content-type of `text/{anything}` will be created as a Zope Page Template. Other kinds of objects, such as images, will continue to be created as specified by the default policy. Note that different `PUT_factories` may be created in different folders; each folder (and its subfolders) will inherit the policy of its particular `PUT_factory`.

To learn more about creating your own custom `PUT_factories`, consult the `PUT_factory` specification .

Using The External Editor Product

Casey Duncan has created a useful Zope product named External Editor. It allows Zope users to use their browsers to navigate the Zope object hierarchy using the ZMI, launching the editor of their choice (for example, vim, Emacs, or Dreamweaver) for that particular object when you click a "pencil" icon next to the object's name. It has both a "client" component, which installs on your local machine and a "server" component, which installs on the server from which you run Zope.

External Editor offers a bit of the "best of both worlds" when it comes to editing Zope content. You can use your existing text and HTML editing tools, manipulated by navigating the Zope Management Interface. It is available for download in Casey's Zope.org area .



Figure 5-2 A Zope With External Editor Installed

Other Integration Facilities

This chapter focused on FTP, DAV, and External Editor. These are the most popular and mature approaches for integration. However, other choices are available.

For instance, Zope has long supported the use of HTTP PUT, originally implemented by Netscape as "Netscape Publishing". This allows Netscape Composer, Mozilla Composer, and Amaya to edit and create new pages, along with associated elements such as images and stylesheets.

Also, Zope provides command-line tools such as `load_site` that can interact with your Zope server.

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 .

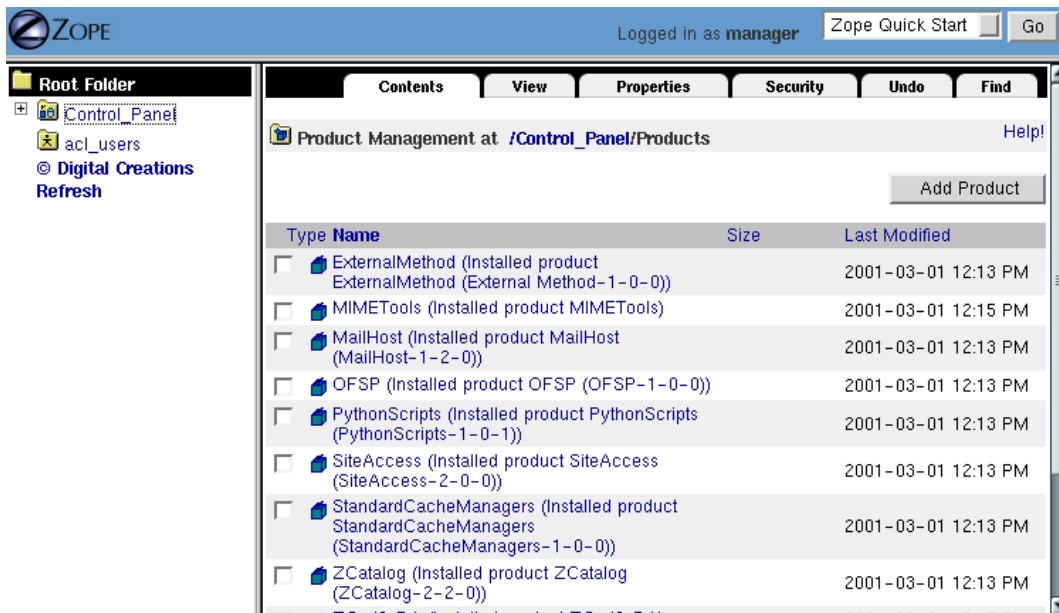


Figure 12-1 Installed Products

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 manageable 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 accomplish 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>
```

Anonymous User - Nov. 20, 2002 9:55 am:

2. If you select the add list, you will see some new types of objects you can add including *ZClass* , *Zope Factory* , and *Zope Permission* .

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 .

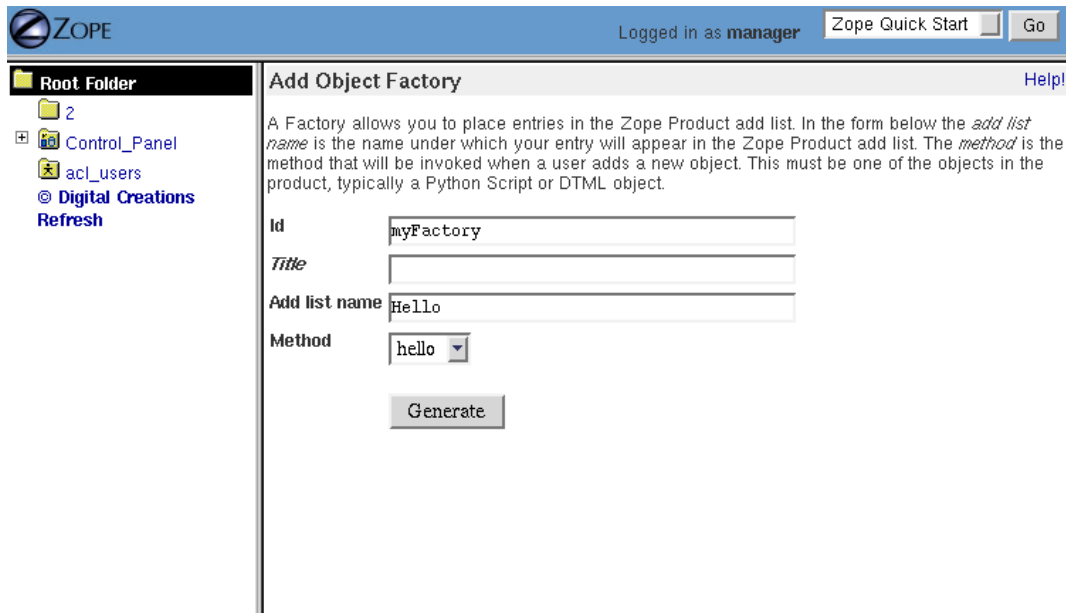


Figure 12-2 Adding A Factory

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')
```

rmg - Sep. 18, 2002 2:36 pm:

The instructions are to 'Paste' in the code. If you paste in with the first three '##' lines, you get an error. Remove those lines and set the parameters in the 'Parameters' form field and all works fine.

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 may 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.

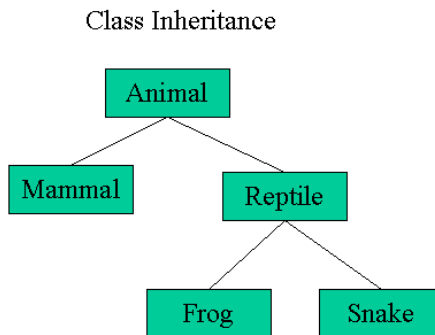


Figure 12-3 Example Class Inheritance

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 .

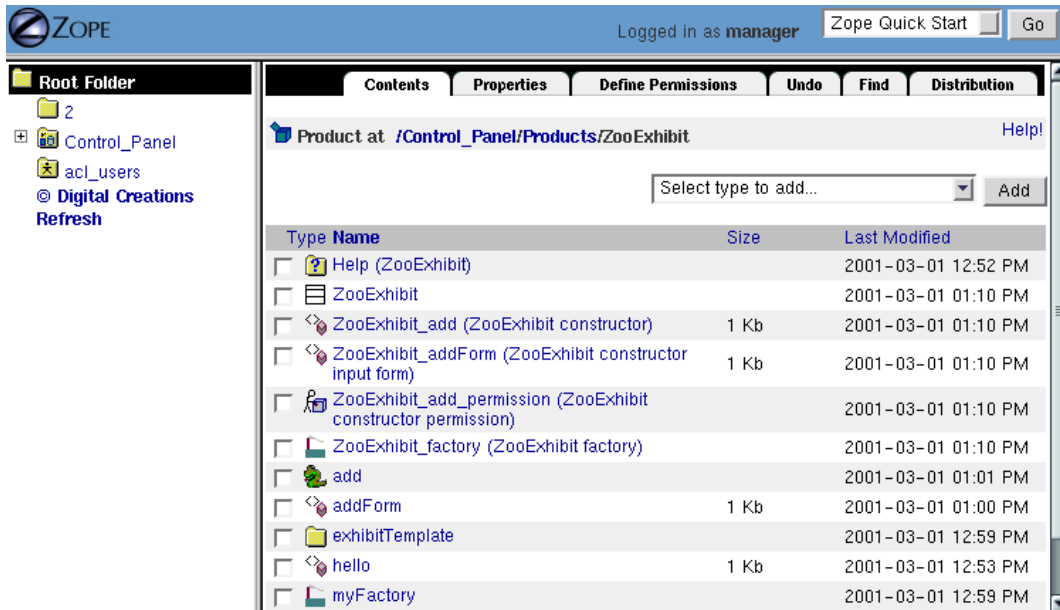


Figure 12-4 Product with a ZClass

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.

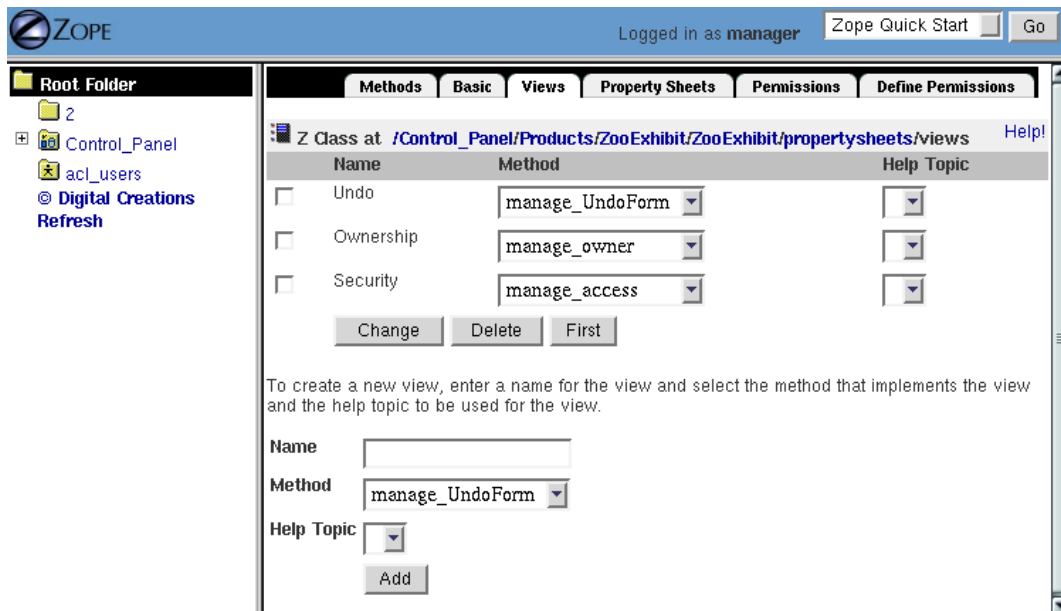


Figure 12-5 The Views view.

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.

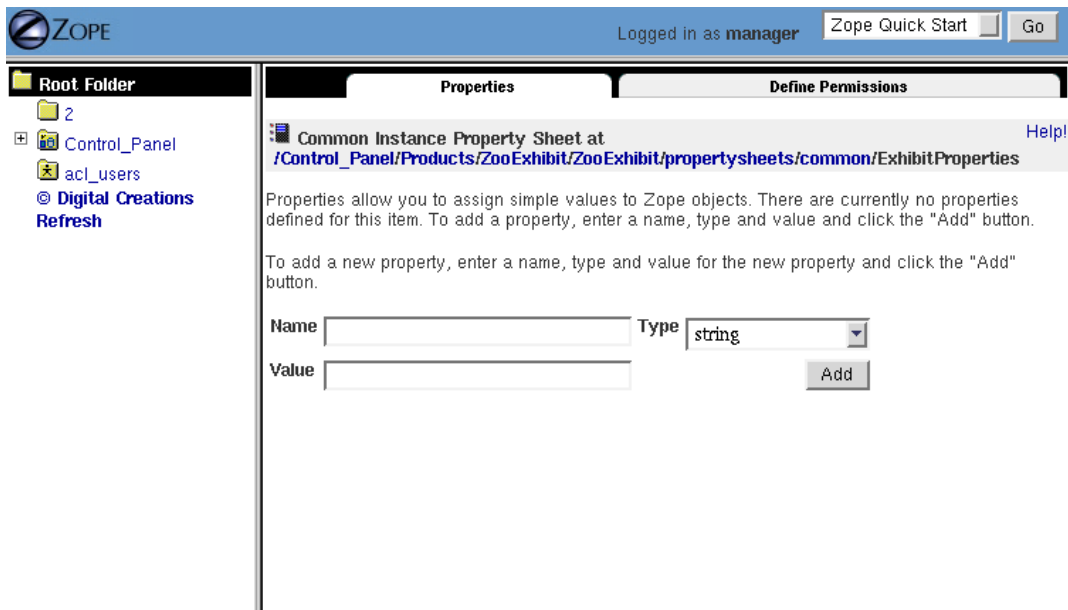


Figure 12-6 A Property Sheet

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 *propertiesheets/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 .

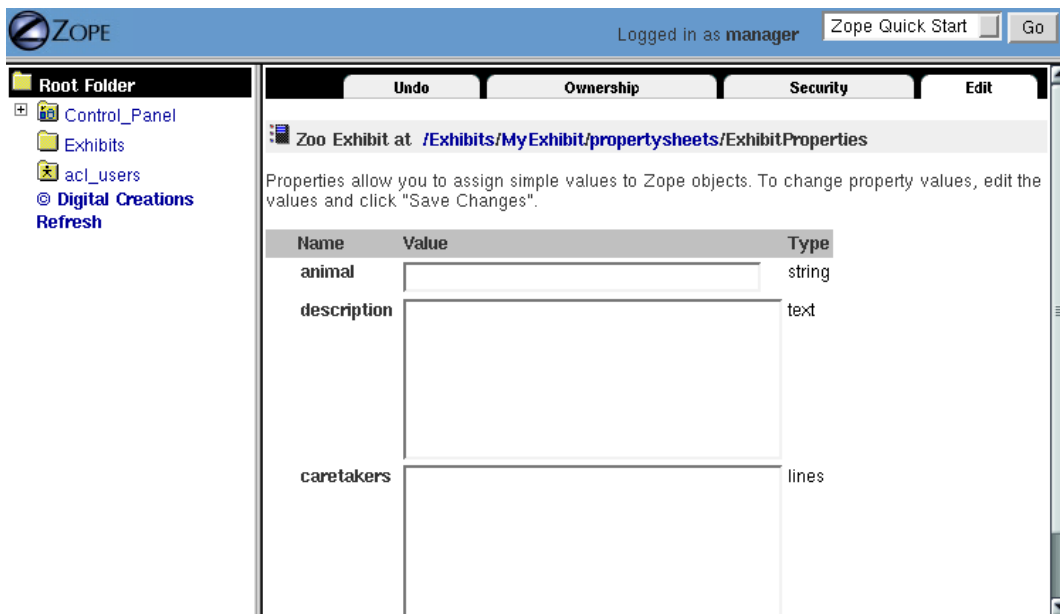


Figure 12-7 A ZooExhibit Edit view

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 .

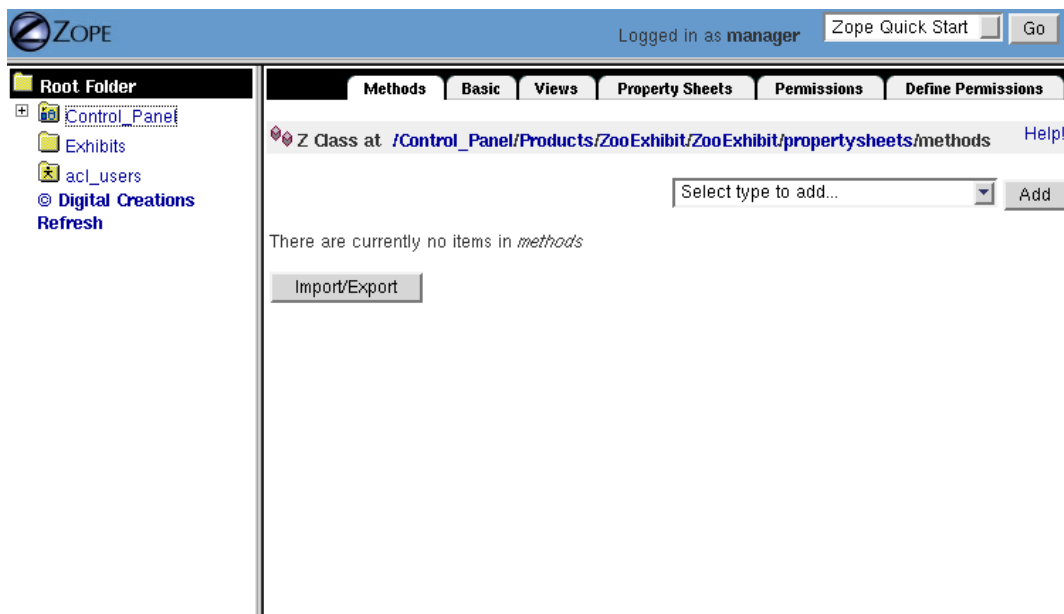


Figure 12-8 The Methods View

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 *showAnimallimages* , and a view called *Images* , you could associate the *showAnimallimages* method with the *Images* view. Whenever anyone clicked on the *Images* view on an instance of your ZClass, the *showAnimallimages* 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 .

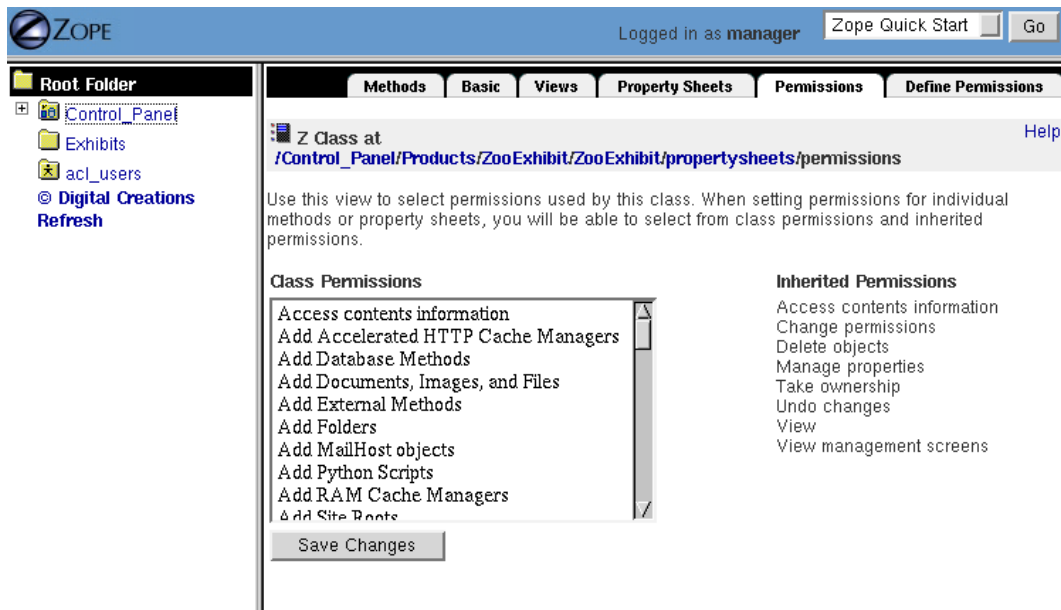


Figure 12-9 The Permissions view

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
```

Anonymous User - Sep. 2, 2002 10:34 am:

```
Let's suppose I need an __init__ method for class Animal: I found that this __init__ method, if present, is
automatically called when I create an instance of
the ZClass that "wraps" (inherits from) Animal. But what if __init__ has parameters? Where can I pass these
parameters from? I guess the right place is from the Animal_add dtml method that acts as a constructor, but
what's the dtml for doing so? Please help!
```

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.

Maintaining Zope

Keeping a Zope site running smoothly involves a number of administrative tasks. This chapter covers some of these tasks, such as:

- Starting Zope automatically at boot time
- Installing new products
- Setting parameters in the Control Panel
- Monitoring
- Cleaning up log files
- Packing and backing up the database
- Database recovery tools

Maintenance often is a very platform-specific task, and Zope runs on many platforms, so you will find instructions for several different operating systems here. It is not possible to provide specifics for every system; instead, we will supply general instructions which should be modified according to your specific needs and platform.

Starting Zope Automatically at Boot Time

For testing and developing purposes you will start Zope manually most of the time, but for production systems it is necessary to start Zope automatically at boot time. Also, we will want to shut down Zope in an orderly fashion when the system goes down. We will describe the necessary steps for Microsoft Windows and some Linux distributions. Take a look at the Linux section for other Unix-like operating systems. Much of the information presented here also applies to System V like Unices.

Debug Mode and Automatic Startup

If you are planning to run Zope on a Unix production system you should also disable *debug mode*. This means removing the `-D` option in startup scripts (e.g. the `start` script created by Zope at installation time which calls `z2.py` with the `-D` switch) and if you've manually set it, unsetting the `Z_DEBUG_MODE` environment variable. In debug mode, Zope does not detach itself from the terminal, which could cause startup scripts to malfunction.

On Windows, running Zope as a service disables debug mode by default. You still can run Zope in debug mode by setting the `Z_DEBUG_MODE` environment variable or running Zope manually from a startup script with the `-D` option. Again, this is not recommended for production systems, since debug mode causes performance loss.

Linux

Distributions with Prepackaged Zope

For many Linux distributions there are ready-made Zope packages which integrate nicely with the system. For instance, Debian, SuSE, Mandrake and Gentoo usually come with fairly recent Zope packages. Those packages contain ready-made scripts for automatic startup.

Automatic Startup for Custom-Built Zopes

Even if you do not want to use the prepackaged Zope that comes with your distribution it should be possible to re-use those startup scripts, eg. by installing the prepackaged Zope and editing the appropriate files and symlinks in `/etc/rc.d` or by extracting them with a tool like `rpm2cpio` .

In the following examples we assume you installed your custom Zope to a system-wide directory, eg. `/usr/local/zope` . If this is not the case please replace every occurrence of `/usr/local/zope` below with your Zope installation directory. There should also be a separate Zope system user present. Below we assume that there is a user `zope` , group `nogroup` present on your system. The user `zope` should of course have read access to the `$ZCOPE_HOME` directory (the directory which contains the "top-level" Zope software and the "z2.py" script) and its descendants, and write access to the contents of the `var` directory.

If you start Zope as root, which is usually the case when starting Zope automatically on system boot, it is required that the `var` directory belongs to root. Set the ownership by executing the command `chown root var` as root.

As an example, the startup script that comes as a part of the SuSE Linux distribution looks like this:

```
#!/bin/sh
# Copyright (c) 1995-2000 SuSE GmbH Nuernberg, Germany.
#
# Authors: Kurt Garloff, Vladimír Linek
#         <feedback@suse.de>
#
# init.d/zope
#
#   and symbolic its link
#
# /usr/sbin/rczope
#
# System startup script for the Zope server
#
### BEGIN INIT INFO
# Provides: zope
# Required-Start: $remote_fs
# Required-Stop: $remote_fs
# Default-Start: 3 5
# Default-Stop: 0 1 2 6
# Description: Start Zope server.
### END INIT INFO

# Source Zope relevant things
. /etc/sysconfig/zope
. /etc/sysconfig/apache

PYTHON_BIN="/usr/bin/python2.1"
test -x $PYTHON_BIN || exit 5
ZCOPE_HOME="/opt/zope"
test -d $ZCOPE_HOME || exit 5
ZCOPE_SCRIPT="$ZCOPE_HOME/z2.py"
test -f $ZCOPE_SCRIPT || exit 5

# Shell functions sourced from /etc/rc.status:
#   rc_check      check and set local and overall rc status
#   rc_status     check and set local and overall rc status
#   rc_status -v  ditto but be verbose in local rc status
#   rc_status -v -r ditto and clear the local rc status
#   rc_failed     set local and overall rc status to failed
#   rc_failed <num> set local and overall rc status to <num><num>
#   rc_reset      clear local rc status (overall remains)
#   rc_exit       exit appropriate to overall rc status
. /etc/rc.status

# First reset status of this service
```

The Zope Book (2.6 Edition)

```
rc_reset

# Return values acc. to LSB for all commands but status:
# 0 - success
# 1 - generic or unspecified error
# 2 - invalid or excess argument(s)
# 3 - unimplemented feature (e.g. "reload")
# 4 - insufficient privilege
# 5 - program is not installed
# 6 - program is not configured
# 7 - program is not running
#
# Note that starting an already running service, stopping
# or restarting a not-running service as well as the restart
# with force-reload (in case signalling is not supported) are
# considered a success.

COMMON_PARAMS="-u zope -z $ZOOPE_HOME -Z /var/run/zope.pid -l /var/log/zope.log"
PCGI_PARAMS="-p $ZOOPE_HOME/Zope.cgi"

[ -z "$ZOOPE_HTTP_PORT" ] && ZOOPE_HTTP_PORT="8080"
ALONE_PARAMS="-w $ZOOPE_HTTP_PORT"

# For debugging...
#SPECIAL_PARAMS="-D"

[ -z "$ZOOPE_FTP_PORT" ] && ZOOPE_FTP_PORT="8021"
if [ "$ZOOPE_FTP" == "yes" ]; then
    SPECIAL_PARAMS="-f $ZOOPE_FTP_PORT $SPECIAL_PARAMS"
fi

if [ "$ZOOPE_PCGI" == "yes" ]; then
    PARAMS="$SPECIAL_PARAMS $PCGI_PARAMS $COMMON_PARAMS"
else
    PARAMS="$SPECIAL_PARAMS $ALONE_PARAMS $COMMON_PARAMS"
fi

case "$1" in
start)
    echo -n "Starting zope"
    ## Start daemon with startproc(8). If this fails
    ## the echo return value is set appropriate.

    # NOTE: startproc return 0, even if service is
    # already running to match LSB spec.
    startproc $PYTHON_BIN $ZOOPE_SCRIPT -X $PARAMS

    # Remember status and be verbose
    rc_status -v
    ;;
stop)
    echo -n "Shutting down zope"
    ## Stop daemon with killproc(8) and if this fails
    ## set echo the echo return value.

    killproc -g -p /var/run/zope.pid -TERM $PYTHON_BIN

    # Remember status and be verbose
    rc_status -v
    ;;
try-restart)
    ## Stop the service and if this succeeds (i.e. the
    ## service was running before), start it again.
    ## Note: try-restart is not (yet) part of LSB (as of 0.7.5)
    $0 status >/dev/null && $0 restart

    # Remember status and be quiet
    rc_status
    ;;
restart)
    ## Stop the service and regardless of whether it was
    ## running or not, start it again.
```

The Zope Book (2.6 Edition)

```
$0 stop
$0 start

# Remember status and be quiet
rc_status
;;
force-reload)
## Signal the daemon to reload its config. Most daemons
## do this on signal 1 (SIGHUP).
## If it does not support it, restart.

echo -n "Reload service zope"
$0 stop && $0 start
rc_status
;;
reload)
## Like force-reload, but if daemon does not support
## signalling, do nothing (!)

rc_failed 3
rc_status -v
;;
status)
echo -n "Checking for zope: "
## Check status with checkproc(8), if process is running
## checkproc will return with exit status 0.

# Status has a slightly different for the status command:
# 0 - service running
# 1 - service dead, but /var/run/ pid file exists
# 2 - service dead, but /var/lock/ lock file exists
# 3 - service not running

# NOTE: checkproc returns LSB compliant status values.
checkproc -p /var/run/zope.pid $PYTHON_BIN
rc_status -v
;;
probe)
## Optional: Probe for the necessity of a reload,
## give out the argument which is required for a reload.

test $ZOPE_HOME/superuser -nt /var/run/zope.pid && echo reload
;;
*)
echo "Usage: $0 {start|stop|status|try-restart|restart|force-reload|reload|probe}"
exit 1
;;
esac
rc_exit
```

You could adapt this script to start your own Zope by modifying the `PYTHON_BIN`, `ZOPE_HOME` and `COMMON_PARAMS`.

To set up a Zope binary package with built-in python situated in `/usr/local/zope` running as user `zope`, with a "WebDAV Source port" set to 8081, you would set:

```
ZOPE_HOME=/usr/local/zope
PYTHON_BIN=$ZOPE_HOME/bin/python
COMMON_PARAMS="-u zope -z $ZOPE_HOME -Z /var/run/zope.pid \
-l /var/log/Z2.log -W 8081 "
```

You can also set up a file `/etc/sysconfig/zope` with variables `ZOPE_FTP_PORT`, `ZOPE_HTTP_PORT`:

```
ZOPE_HTTP_PORT=80
ZOPE_FTP_PORT=21
```

to set the HTTP and FTP ports. The default is to start them at port 8080 and 8021.

For Red Hat, you can find a start-up script which seems to work here

Unfortunately, all Linux distributions start and stop services a little differently, so it is not possible to write a startup script that integrates well with every distribution. We will try to outline a crude version of a generic startup script which you can refine according to your needs.

To do this some shell scripting knowledge and root system access is required.

Linux startup scripts usually reside in `/etc/init.d` or in `/etc/rc.d/init.d`. For our examples we assume the startup scripts to be in `/etc/rc.d/init.d`, adjust if necessary.

To let the boot process call a startup script, you also have to place a symbolic link to the startup script in the `/etc/rc.d/rc?.d` directories, where '?' is a number from 0-6 which stands for the SystemV run levels. You usually will want to start Zope in run levels 3 and 5 (3 is full multi-user mode, 5 is multiuser mode with X started, according to the "Linux Standard Base":<http://www.linuxbase.org>), so you would place two links in the `/etc/rc.d` directories. Be warned that some systems (such as Debian) assume that runlevel 2 is full multiuser mode. As stated above, we assume the main startup script to be located in `/etc/rc.d/init.d/zope`, if your system puts the `init.d` directory somewhere else, you should accommodate the paths below:

```
# cd /etc/rc.d/rc3.d
# ln -s /etc/rc.d/init.d/zope S99zope
# cd /etc/rc.d/rc5.d
# ln -s /etc/rc.d/init.d/zope S99zope
```

The scripts are called by the boot process with an argument `start` when starting up and `stop` on shutdown.

A simple generic startup script structure could be something like this:

```
#!/bin/sh

# set paths and startup options
ZOPE_HOME=/usr/local/zope
PYTHON_BIN=$ZOPE_HOME/bin/python
ZOPE_OPTS="-u zope -P 8000"
EVENT_LOG_FILE=$ZOPE_HOME/var/event.log
EVENT_LOG_SEVERITY=-300
# define more environment variables ...

export EVENT_LOG_FILE EVENT_LOG_SEVERITY
# export more environment variables ...

umask 077
cd $ZOPE_HOME

case "$1" in

start)
    # start service
    exec $PYTHON_BIN $ZOPE_HOME/z2.py $ZOPE_OPTS

    # if you want to start in debug mode (not recommended for
    # production systems):
    # exec $PYTHON_BIN $ZOPE_HOME/z2.py $ZOPE_OPTS -D &
    ;;

stop)
    # stop service
    kill `cat $ZOPE_HOME/var/Z2.pid`
    ;;

restart)
    # stop service and restart
    $0 stop
    $0 start
    ;;

;;
```

```
*)
echo "Usage: $0 {start|stop|restart}"
exit 1
;;
esac
```

This script lets you perform start / stop / restart operations:

start — Start Zope (and the zdaemon management process)

stop — Stop Zope. Kill Zope and the zdaemon management process

restart — Stop then start Zope

Mac OS X

For Mac OS X, there is a website:<http://zope-mosx.zopeonarope.com> devoted to running Zope on Mac OS X; you also might want to look here:http://www.zope.org/Members/jens/docs/zope_osx for building / installing Zope on OS X. You might also want to check out this:http://www.zope.org/Members/richard/zope_controller/mac for starting / stopping Zope on Mac OS X, although it is currently unmaintained.

MS Windows

The prevalent way to autostart Zope on MS Windows is to install Zope as a service. However, since services are not available on Windows 95, Windows 98, or Windows ME, you will have to resort to the somewhat crude method of putting a link to Zope's `start.bat` script into the `Startup` folder of the Windows start menu on those platforms.

If you installed Zope on Windows NT/2000/XP to be started manually and later on want it started as a service, perform these steps from the command line to register Zope as a Windows service:

```
> cd c:\Program Files\zope
> bin\lib\win32\PythonService.exe /register
> bin\python.exe ZServer\ZService.py --startup auto install
```

Replace `c:\Program Files\zope` with the path to your Zope installation. Zope should now be installed as a service which starts automatically on system boot. To start and stop Zope manually, go to the Windows service administration tool, right-click the Zope service and select the corresponding entry.

Installing New Products

Zope is a framework for building websites from new and existing software, known as Zope *products*. A product is a Python package with special conventions that register with the Zope framework. The primary purpose of a Zope product is to create new kinds of objects that appear in the add list. This extensibility through products has spawned a broad market of add-on software for Zope.

The guidelines for packaging a product are given in the "Packaging Products" section in the Zope Products chapter of the Zope Developer Guide. However, since these guidelines are not enforced, many Zope products adhere to different conventions. This section will discuss the different approaches to installing Zope packages.

To install a Zope product, you first download an archive file from a website, such as the Downloads section of zope.org. These archive files come in several varieties, such as `tgz` (gzipped tar files), `zip` (the popular ZIP format common on Windows), and others.

In general, unpacking these archives will create a subdirectory containing the Product itself. For instance, the `Poll-1.0.tgz` archive file in the "Packaging Products" section mentioned above contains a subdirectory of `Poll` . All the software is contained in this directory.

To install the product, you unarchive the file in the `lib/python/Products` directory. In the Poll example, this will create a directory `lib/python/Products/Poll` .

Unfortunately not all Zope developers adhere to this convention. Often the archive file will have the `lib/python/Products` part of the path included. Worse, the archive might contain no directory, and instead have all the files in the top-level of the archive. Thus, it is advised to inspect the contents of the archive first.

Once you have the new directory in `lib/python/Products` , you need to tell Zope that a new product has been added. You can do this by restarting your Zope server through the Control Panel of the Zope Management Interface (ZMI), or, on POSIX systems, by sending the Zope process a `-HUP` signal. For instance, from the Zope directory:

```
kill -HUP `cat var/Z2.pid`
```

If your Zope server is running in debug mode, a log message will appear indicating a new product has been discovered and registered.

To confirm that your product is installed, log into your Zope site and visit the Control Panel's Products section. You should see the new product appear in the list of installed products.

If there was a problem with the installation, the Control Panel will list it as a "Broken Product". Usually this is because Python had a problem importing a package, or the software had a syntax error. You can visit the broken product in the Control Panel and click on its *Traceback* tab. You will see the Python traceback generated when the package was imported.

A traceback generally will tell you what went wrong with the import. For instance, a package the software depends on could be missing. To illustrate this take a look at the traceback below - a result of trying to install `CMFOODocument`:<http://www.zope.org/Members/longsleep/CMFOODocument> without the (required) `CMF` package:

```
Traceback (most recent call last):
File "/usr/share/zope/2.6.0/lib/python/OFS/Application.py", line 541, in import_product
product=__import__(pname, global_dict, global_dict, silly)
File "/usr/share/zope/2.6.0/lib/python/Products/CMFOODocument/__init__.py", line 19, in ?
import OODocument
File "/usr/share/zope/2.6.0/lib/python/Products/CMFOODocument/OODocument.py", line 31, in ?
from Products.CMFCore.PortalContent import NoWL, ResourceLockedError
ImportError: No module named CMFCore.PortalContent
```

Server Settings

The Zope server has a number of settings that can be adjusted for performance. Unfortunately, performance tuning is not an exact science, that is, there is no recipe for setting parameters. Rather, you have to test every change. To load test a site, you should run a test setup with easily reproducible results. Load test a few significant spots in your application. The trick is to identify typical situations while still permitting automated testing. There are several tools to load test websites. One of the simple yet surprisingly useful tools is `ab` which comes with Apache distributions. With `ab` you can test individual URLs, optionally providing cookies and POST data. Other tools often allow one to create or record a user session and playing it back multiple times. See eg. the Open System Testing Architecture , `JMeter` , or Microsoft's Web Application Stress Tool .

Database Cache

The most important is the database cache setting. To adjust these settings, visit the Control Panel and click on the *Database* link.

IMAGE UNAVAILABLE

Figure 23-1 Database Cache settings

There are usually seven database connections to the internal Zope database (see *Database Connections* below for information about how to change the number of connections). Each connection gets its own database cache. The "Target number of objects in memory per cache" setting controls just that - the system will try not to put more than this number of persistent Zope objects into RAM per database connection. So if this number is set to 400 and there are seven database connections configured, there should not be more than 2800 objects sitting in memory. Obviously, this does not say much about memory consumption, since the objects might be anything in size - from a few hundred bytes upwards. The cache favors commonly used objects - it wholly depends on your application and the kind of objects which memory consumption will result from the number set here. As a rule, Zope objects are about as big as the data they contain. There is only little overhead in wrapping data into Zope objects.

Note that only objects residing in the Zope object database are affected - data residing in external files (for instance through the ExtFile or LocalFS add-on products) or in relational databases connected to Zope will not be cached here.

Interpreter Check Intervals

The interpreter check interval determines how often the interpreter stops to execute Zope code and checks for housekeeping things like signal handlers and thread switches. A higher number means it stops less often. The default of 500 should give good performance with most platforms, but you may want to experiment with other values. The general rule is to set it higher for faster machines, so if you have a really fast system you could try setting this higher.

The interpreter check interval is set with the `-i` argument to `z2.py` script. This means you should set it in Zopes own start script if you start Zope manually, or in the system start script.

ZServer Threads

This number determines how many ZServer threads Zope starts to service requests. The default number is four (4). You may try to increase this number if you are running a heavily loaded website. If you want to increase this to more

than seven (7) threads, you also should increase the number of database connections (see the next section).

Database Connections

We briefly mentioned Zope's internal database connections in the *Database Cache* section above. Out of the box, the number of database connections is hardwired to seven (7); but this can be changed. There is no "knob" to change this number so in order to change the number of database connections, you will need to enter quite deep into the systems' bowels. It is probably a wise idea to back up your Zope installation before following any of the instructions below.

Each database connection maintains its own cache (see above, "Database Cache"), so bumping the number of connections up increases memory requirements. Only change this setting if you're sure you have the memory to spare.

To change this setting, create a file called "custom_zodb.py" in your Zope installation directory. In this file, put the following code:

```
import ZODB.FileStorage
import ZODB.DB

filename = os.path.join(INSTANCE_HOME, 'var', 'Data.fs')
Storage = ZODB.FileStorage.FileStorage(filename)
DB = ZODB.DB(Storage, pool_size=25, cache_size=2000)
```

This only applies if you are using the standard Zope FileStorage storage.

The "pool_size" parameter is the number of database connections. Note that the number of database connections should always be higher than the number of ZServer threads by a few (it doesn't make sense to have fewer database connections than threads). See above on how to change the number of ZServer threads.

Signals (POSIX only)

Signals are a POSIX inter-process communications mechanism. If you are using Windows then this documentation does not apply.

Zope responds to signals which are sent to the process id specified in the file '\$ZCOPE_HOME/var/Z2.pid':

```
SIGHUP - close open database connections, then restart the server
        process. The common idiom for restarting a Zope server is:

        kill -HUP `cat $ZCOPE_HOME/var/Z2.pid`

SIGTERM - close open database connections then shut down. The common
         idiom for shutting down Zope is:

        kill -TERM `cat $ZCOPE_HOME/var/Z2.pid`

SIGINT - same as SIGTERM

SIGUSR2 - close and re-open all Zope log files (z2.log, event log,
         detailed log.) The common idiom after rotating Zope log files
         is:

        kill -USR2 `cat $ZCOPE_HOME/var/Z2.pid`
```

The process id written to the `Z2.pid` file depends on whether Zope is run under the `zdaemon` management process. If Zope is run under a management process (as it is by default) then the pid of the management process is recorded here. Relevant signals sent to the management process are forwarded on to the server process. Specifically, it forwards all those signals listed above, plus `SIGQUIT` and `SIGUSR1`. If Zope is not using a management process (`-Z0` on the `z2.py` command line), the server process records its own pid into `z2.pid`, but all signals work the same way.

Monitoring

To detect problems (both present and future) when running Zope on production systems, it is wise to watch a few parameters.

Monitor the Event Log and the Access Log

If you set the `EVENT_LOG_FILE` (formerly known as the `STUPID_LOG_FILE`) as an environment variable or a parameter to the startup script, you can find potential problems logged to the file set there. Each log entry is tagged with a severity level, ranging from `TRACE` (lowest) to `PANIC` (highest). You can set the verbosity of the event log with the environment variable `EVENT_LOG_SEVERITY`. You have to set this to an integer value - see below:

```
TRACE=-300  -- Trace messages
DEBUG=-200  -- Debugging messages
BLATHER=-100 -- Somebody shut this app up.
INFO=0      -- For things like startup and shutdown.
PROBLEM=100 -- This isn't causing any immediate problems, but
deserves attention.
WARNING=100 -- A wishy-washy alias for PROBLEM.
ERROR=200   -- This is going to have adverse effects.
PANIC=300   -- We're dead!
```

So, for example setting `EVENT_LOG_SEVERITY=-300` should give you all log messages for Zope and Zope applications that use Zopes' logging system.

You also should look at your access log (usually placed in `$ZCOPE_HOME/var/Z2.log`). The `Z2.log` file is recorded in the Common Log Format . The sixth field of each line contains the HTTP status code. Look out for status codes of `5xx`, server error. Server errors often point to performance problems.

Monitor the HTTP Service

You can find several tools on the net which facilitate monitoring of remote services, for example Nagios or VisualPulse .

For a simple "ping" type of HTTP monitoring, you could also try to put a small DTML Method with a known value on your server, for instance only containing the character "1". Then, using something along the line of the shell script below, you could periodically request the URL of this DTML Method, and mail an error report if we are getting some other value (note the script below requires a Un*x-like operating system):

```
#!/bin/sh

# configure the values below
URL="http://localhost/ping"
EXPECTED_ANSWER="1"
MAILTO="your.mailaddress@domain.name"
SUBJECT="There seems to be a problem with your website"
MAIL_BIN="/bin/mail"

resp=`wget -O - -q -t 1 -T 1 $URL`
if [ "$resp" != "$EXPECTED_ANSWER" ]; then
  $MAIL_BIN -s "$SUBJECT" $MAILTO <<EOF
The URL
-----
$URL
-----
```

```
did not respond with the expected value of $EXPECTED_ANSWER.  
EOF  
fi;
```

Run this script eg. every 10 minutes from cron and you should be set for simple tasks. Be aware though that we do not handle connections timeouts well here. If the connection hangs, for instance because of firewall misconfiguration, `wget` will likely wait for quite a while (around 15 minutes) before it reports an error.

Log Files

There are two main sources of log information in Zope, the access log and the event log.

Access Log

The access log records every request made to the HTTP server. It is recorded in the Common Log Format .

The default target of the access log is the file `$ZCOPE_HOME/var/Z2.log`. Under Unix it is however possible to direct this to the syslog by setting the environment variable `ZSYSLOG_ACCESS` to the desired domain socket (usually `/dev/log`)

If you are using syslog, you can also set a facility name by setting the environment variable `ZSYSLOG_FACILITY`. It is also possible to log to a remote machine. This is also controlled, you might have guessed it, by an environment variable. The variable is called `ZSYSLOG_SERVER` and should be set to a string of the form "host:port" where host is the remote logging machine name or IP address and port is the port number the syslog daemon is listening on (usually 514).

Event Log

The event log (formerly also called "stupid log") logs Zope and third-party application message. The ordinary log method is to log to a file specified by the `EVENT_LOG_FILE`, eg. `EVENT_LOG_FILE=$ZCOPE_HOME/var/event.log`

On Unix it is also possible to use the syslog daemon by setting the environment variable `ZSYSLOG` to the desired Unix domain socket, usually `/dev/log` . Like with access logs (see above), it is possible to set a facility name by setting the `ZSYSLOG_FACILITY` environment variable, and to log to a remote logging machine by setting the `ZSYSLOG_SERVER` variable to a string of the form "host:port", where port usually should be 514.

You can coarsely control how much logging information you want to get by setting the variable `EVENT_LOG_SEVERITY` to an integer number - see the section "Monitor the Event Log and the Access Log" above.

Log Rotation

Log files always grow, so it is customary to periodically rotate logs. This means logfiles are closed, renamed (and optionally compressed) and new logfiles get created. On Unix, there is the `logrotate` package which traditionally handles this. A sample configuration might look like this:

```
compress  
/usr/local/zope/var/Z2.log {  
    rotate 25  
    weekly  
    postrotate  
        /sbin/kill -USR2 `cat /usr/local/zope/var/Z2.pid`  
    endsript  
}
```

This would tell logrotate to compress all log files (not just Zope's!), handle Zopes access log file, keep 25 rotated log files, do a log rotation every week, and send the SIGUSR2 signal to Zope after rotation. This will cause Zope to close the logfile and start a new one. See the documentation to `logrotate` for further details.

On Windows there are no widespread tools for log rotation. You might try the KiWi Syslog Daemon and configure Zope to log to it. Also see the sections "Access Log" and "Event Log" above.

Packing and Backing Up the FileStorage Database

The storage used by default by Zope's built-in object database, FileStorage, is an undoable storage. This essentially means changes to Zope objects do not overwrite the old object data, rather the new object gets appended to the database. This makes it possible to recreate an objects previous state, but it also means that the file the objects are kept in (which usually resides in `$ZOPE_HOME/var/Data.fs`) always keeps growing.

To get rid of obsolete objects, you need to `pack` the ZODB. This can be done manually by opening Zopes Control_Panel and clicking on the "Database Management" link. Zope offers you the option of removing only object version older than an adjustable amount of days.

If you want to automatically pack the ZODB you could tickle the appropriate URL with a small python script (the traditional filesystem based kind, not Zopes "Script (Python)"):

```
#!/usr/bin/python
import sys, urllib
host = sys.argv[1]
days = sys.argv[2]
url = "%s/Control_Panel/Database/manage_pack?days:float=%s" % \
    (host, days)
try:
    f = urllib.urlopen(url).read()
except IOError:
    print "Cannot open URL %s, aborting" % url
print "Successfully packed ZODB on host %s" % host
```

The script takes two arguments, the URL of your server (eg. `http://mymachine.com`) and the number of days old an object version has to be to get discarded.

On Unix, put this in eg. the file `/usr/local/sbin/zope_pack` , and make it executable with `chmod +x zope_pack` . Then you can put in into your crontab with eg.:

```
5 4 * * sun    /usr/local/sbin/zope_pack http://localhost 7
```

This would instruct your system to pack the ZODB on 4:05 every sunday. It would connect to the local machine, and leave object versions younger than 7 days in the ZODB.

Under Windows, you should use the scheduler to periodically start the script. Put the above script in eg. `c:\Program Files\zope_pack.py` or wherever you keep custom scripts, and create a batch file `zope_pack.bat` with contents similar to the following:

```
"C:\Program Files\zope\bin\python.exe" "C:\Program Files\zope_pack.py" "http://localhost" 7
```

The first parameter to python is the path to the python script we just created. The second is the root URL of the machine you want to pack, and the third is the maximum age of object versions you want to keep. Now instruct the scheduler to run this `.bat` file every week.

Zope backup is quite straightforward. If you are using the default storage (FileStorage), all you need to do is to save the file `$ZOPE_HOME/var/Data.fs` . This can be done online, because Zope only appends to the `Data.fs` file - and if a few bytes are missing at the end of the file due to a copy while the file is being written to, ZODB is usually capable of

repairing that upon startup. The only thing to worry about would be if someone were to be using the *Undo* feature during backup. If you cannot ensure that this does not happen, you should take one of two routes. The first is to be to shutdown Zope prior to a backup, and the second is to do a packing operation in combination with backup. Packing the ZODB leaves a file `Data.fs.old` with the previous contents of the ZODB. Since Zope does not write to that file anymore after packing, it is safe to backup this file even if undo operations are performed on the live ZODB.

To backup `Data.fs` on Linux, you should not `tar` it directly, because `tar` will exit with an error if files change in the middle of a `tar` operation. Simply copying it over first will do the trick. Another option is to use `rsync` like in this shell script by Jeff Rush:

```
#!/bin/sh
#####
# File: /etc/cron.daily/zbackup.cron
#
# Backup Zope Database Daily
#####
#
# rsync arguments:
# -q      := Quiet operation, for cron logs
# -u      := Update only, don't overwrite newer files
# -t      := Preserve file timestamps
# -p      := Preserve file permissions
# -o      := Preserve file owner
# -g      := Preserve file group
# -z      := Compress during transfer
# -e ssh  := Use the ssh utility to secure the link
#
ARCHTOP="/archive/zope/"
DOW=`date +%A`
ARCHDIR="${ARCHTOP}${DOW}"
#
# Insure Our Day-of-Week Directory Exists
[ -d ${ARCHDIR} ] || mkdir ${ARCHDIR} || {
    echo "Could Not Create Day-of-Week Directory: ${ARCHDIR}" ; exit 1
}
#
/usr/bin/rsync -q -u -t -p -o -g /var/zope/var/Data.fs ${ARCHDIR}
#
ln -sf ${ARCHDIR} ${ARCHTOP}Current
#
exit 0
```

This script should be run daily from cron. It will create day-of-week subdirectories under `/archive/zope`, and update the `Data.fs` file there with the current version. `rsync` only transmits *differences* between files, so only a minimal amount of disk copying is needed. This could be advantageous especially for large ZODB databases.

Database Recovery Tools

To recover data from corrupted ZODB database file (typically located in `$ZOPE_HOME/var/Data.fs`) there is a script `fsrecover.py` located in `$ZOPE_HOME/lib/python/ZODB`.

`fsrecover.py` has the following help output:

```
python fsrecover.py [ <options> ] inputfile outputfile
```

Options:

```
-f -- force output even if output file exists
```

```
-v level -- Set the
verbosity level:
```

```
0 -- Show progress indicator (default)
```

```
1 -- Show transaction times and sizes
```

The Zope Book (2.6 Edition)

- 2 -- Show transaction times and sizes, and show object (record) ids, versions, and sizes.
- p -- Copy partial transactions. If a data record in the middle of a transaction is bad, the data up to the bad data are packed. The output record is marked as packed. If this option is not used, transaction with any bad data are skipped.
- P t -- Pack data to t seconds in the past. Note that is the "-p" option is used, then t should be 0.

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 DTML's 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 method's arguments are passed automatically by DTML just as with the `var` tag. If the method is specified in an 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>
```

Anonymous User - Apr. 29, 2002 2:00 pm:

This explanation is misleading and partially incorrect. Zope will not save any comments that are not valid DTML. So you cannot not comment: using `<dtml-in myExample>` to loop ... This produces a Zope error.

This is counter-intuitive and cripples good documentation practices. Zope collector Issue 370

<http://collector.zope.org/Zope/370>

Anonymous User - May 16, 2002 5:53 pm:

Also, it appears (at least in Zope 2.32-ish) that DTML commands inside comments ARE parsed, since errors in syntax or references to objects, methods, etc. that are invalid will cause Zope to complain about the file. Normal programming language syntax would dictate that NOTHING after a start-comment token should be parsed until an end-comment token is encountered.

Anonymous User - July 26, 2002 3:31 pm:

Agreed with the normal prg lang syntax...

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 Figure, 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(*Figure start*, *stop Figure*, *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 Figure*, *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 Figure*, *with Figure*, *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(*innerlink object Figure*, *parent Figure*, *name Figure*, *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*]... *Figure*, *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>
```

```
Anonymous User - May 8, 2002 7:56 am:
a commenting identifier at the end tag is allowed and will be ignored like
</dtml-in my_short_sequ_name>
same for if / let (dunno abt other tags)
```

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 rows:

```
<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-sequence-start-number>">Previous</a>
</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-start-number>">Next</a>
</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-boundary>]
...
</dtml-mime>
```

```
Anonymous User - Apr. 6, 2002 8:09 pm:
/boundary/boundary
```

The `mime` tag is a block tag. The block is can be divided by one or more `boundary` 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 `boundary` 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-mime>
</dtml-sendmail>
```

```
Anonymous User - June 10, 2002 5:47 am:
There's a <dtml-recipient> tag?!?!
I think line 2 of the code is supposed to read:
  To: &dtml-recipient;
```

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>
```

Anonymous User - May 22, 2002 11:37 am:
Looks like the 3rd <dtml-sqlgroup> should be a close tag: </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>
```

```
Anonymous User - June 4, 2002 8:00 am:
Is there a way to prevent the tree of showing the user folder?
And how can I change the order of the displayed folders?
```

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>
```

```
rklahn - Aug. 19, 2002 7:29 pm:
It is often useful to generate a tree of your parent, from a DTML document. I think it would be helpful if an
example was provided that performed this function, such as:
<dtml-tree expr="aq_parent" skip_unauthorized="1">
  <a href="&dtml-absolute_url;"><dtml-var title_or_id></a>
</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-exception>
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>
mcdonc - Aug. 14, 2002 11:47 am:
  Need docs for url_unquote_plus and url_unquote (added in 2.6)
```

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="%0.2f">
```

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 through 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")
```

The Zope Book (2.6 Edition)

```
# 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

AMPMinutes()

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.

getContentTypes()

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 that 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>
    ">
  </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 where the `messageText` is an rfc822 formatted message. This allows you complete control over the message headers, including setting any extra headers such as Cc: and Bcc:. The arguments are:

messageText — The mail message. It can either be a rfc822 formed text with header fields, or just a body without any header fields. The other arguments given will override the header fields in the message, if they exist.

mto — A commaseparated 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`.

```
simple_send(self, mto, mfrom, subject, body)
```

Sends a message. Only To:, From: and Subject: headers can be set. The arguments are:

mto — A commaseparated string or list of recipient(s) of the message.

mfrom — The address of the message sender.

subject — The subject of the message.

body — The body of the message.

MailHost 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)
```

```
Anonymous User - June 13, 2002 4:37 pm:  
what is OFSP?????  
explanation would help here...
```

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>
```

Anonymous User - July 17, 2002 9:30 pm:

The above sample illustrates how to pass a `meta_type` to the `objectValues` method. But how do you pass a set of `meta_types` to `objectValues` method?

Anonymous User - July 17, 2002 9:40 pm:

Use list to specify a set of `meta_types`, try this:

```
<dtml-in expr="objectValues(['File','Folder'])">
  ...
</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` properties 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 separated 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 separated 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](http://hoohoo.ncsa.uiuc.edu/cgi/env.html) in the [CGI specification](http://hoohoo.ncsa.uiuc.edu/cgi/interface.html)

- 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.

URL_n — 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` .

URLPATH_n — 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= `/` .

BASE_n — 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` .

BASEPATH_n — 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 , URL_n , URLPATH_n , BASE_n , BASEPATH_n , 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` Script module

This provides generic script support

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 mutable 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 different 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.

Otherwise, 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'})
```

```
Anonymous User - Aug. 6, 2002 10:03 am:
missing ' in "'author"
```

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 parameter 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 be used 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 creation 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":<http://www.python.org/doc/current/lib/module-math.html> 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":<http://www.python.org/doc/current/lib/module-random.html> 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

Products.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":http://www.python.org/doc/current/lib/Functions_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":<http://www.python.org/doc/current/lib/module-urllib.html> 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":<http://www.python.org/doc/current/lib/module-urllib.html> 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 optional 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":<http://www.python.org/doc/current/lib/module-string.html> 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). It also describes some ZPT-specific behaviors that are not part of the standards.

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.

Due to the fact that TAL sees statements as XML attributes, even in HTML documents, it cannot use the order in which statements are written in the tag to determine the order in which they are executed. TAL must also forbid multiples of the same kind of statement on a single element, so it is sufficient to arrange the kinds of statement in a precedence list.

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-prefix ':' ] Name
namespace-prefix  ::= 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.

The default replacement behavior is `text`, which replaces angle-brackets and ampersands with their HTML entity equivalents. The `structure` keyword passes the replacement text through unchanged, allowing HTML/XML markup to be inserted. This can break your page if the text contains unanticipated markup (eg. text submitted via a web form), which is the reason that it is not the default.

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 an element in place while omitting the surrounding start and end tags.

If the expression evaluates to a *false* value, then normal processing of the element continues and the tags are not omitted. If the expression evaluates to a *true* value, or no expression is provided, the statement element is replaced with its contents.

Zope treats empty strings, empty sequences, zero, None, and *nothing* as false. All other values are considered true, including *default*.

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 occurred.</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"` .

With the exception of `start` , `end` , and `index` , all of the attributes of a repeat variable are methods. Thus, when you use a Python expression to access them, you must call them, as in `"python:repeat['item'].length()"` .

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::

```
<p tal:repeat="txt python:'one', 'two', 'three'">
  <span tal:replace="txt" />
</p>
```

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. Separates 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. positive and negative numbers 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* 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 [ '|' Expression ]
Path        ::= variable [ '/' PathSegment ]*
variable    ::= Name
PathSegment ::= ( '?' variable ) | PathChar+
PathChar    ::= AlphaNumeric | ' ' | '_' | '-' | '.' | ',' | '~'
```

Description

A path expression consists of a *path* optionally followed by a vertical bar (|) and alternate expression. A path consists of one or more non-empty strings separated by slashes. The first string must be a variable name (a 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.

A limited amount of indirection is possible by using a variable name prefixed with ? as a path segment. The variable must contain a string, which replaces that segment before the path is traversed.

For example:

```
request/cookies/oatmeal
nothing
here/some-file_2001_02.html.tar.gz/foo
root/to/branch | default

request/name | string:Anonymous Coward
here/?tname/macros/?mname
```

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, and no alternate expression has been specified, an error results. Otherwise, the alternate expression is evaluated.

The alternate expression can be any TALEX expression. For example, `request/name | string:Anonymous Coward` is a valid path expression. This is useful chiefly for providing default values, such as strings and numbers, which are not expressible as path expressions. Since the alternate expression can be a path expression, it is possible to "chain" path expressions, as in `first | second | third | nothing`.

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 TALEX overview for a list of built-in variables. Variable names are looked up first in locals, then in globals, then in the built-in list, so the built-in variables 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`

ZPT-specific Behaviors

The behavior of Zope Page Templates is almost completely described by the TAL, TALES, and METAL specifications. ZPTs do, however, have a few additional features that are not described in the standards.

HTML Support Features

When the content-type of a Page Template is set to `text/html`, Zope processes the template somewhat differently than with any other content-type. As mentioned under TAL Namespace, HTML documents are not required to declare namespaces, and are provided with `tal` and `metal` namespaces by default.

HTML documents are parsed using a non-XML parser that is somewhat more forgiving of malformed markup. In particular, elements that are often written without closing tags, such as paragraphs and list items, are not treated as errors when written that way, unless they are statement elements. This laxity can cause a confusing error in at least one case; A `<div>` element is block-level, and therefore technically not allowed to be nested in a `<p>` element, so it will cause the paragraph to be implicitly closed. The closing `</p>` tag will then cause a `NestingError`, since it is not matched up with the opening tag. The solution is to use `` instead.

Unclosed statement elements are always treated as errors, so as not to cause subtle errors by trying to infer where the element ends. Elements which normally do not have closing tags in HTML, such as image and input elements, are not required to have a closing tag, or to use the XHTML `<tag />` form.

Certain boolean attributes, such as `checked` and `selected`, are treated differently by `tal:attributes`. The value is treated as true or false (as defined by `tal:condition`). The attribute is set to `attr="attr"` in the true case and omitted otherwise. If the value is `default`, then it is treated as true if the attribute already exists, and false if it does not. For example, each of the following lines:

```
<input type="checkbox" checked tal:attributes="checked default">
<input type="checkbox" tal:attributes="checked string:yes">
<input type="checkbox" tal:attributes="checked python:42">
```

...will render as:

```
<input type="checkbox" checked="checked">
```

...while each of these:

```
<input type="checkbox" tal:attributes="checked default">
<input type="checkbox" tal:attributes="checked string:">
<input type="checkbox" tal:attributes="checked nothing">
```

...will render as:

```
<input type="checkbox">
```

This works correctly in all browsers in which it has been tested.

Appendix D: Zope Resources

At the time of this writing there is a multitude of sources for Zope information on the Internet and 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.

ZopeLabs <http://www.zopelabs.com> is a website dedicated to gathering "recipe-like" snippets of Zope programming logic. It is run by Adam Kendall.

My-Zope is a weblog about Zope run by noted community member "kedai".

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.

(Other) Zope Books

The Zope Bible by Scott Robertson and Michael Bernstein.

The Book of Zope by Beehive.

The Zope Web Application Construction Kit edited by Martina Brockman, et. al.

Zope: Web Application Development and Content Management edited by Steve Spicklemire et al.

Mailing Lists

Zope.org maintains a collection of the many Zope mailing lists.

Python Information

Python.org has lots of information about Python including a tutorial and reference documentation.

DTML Name Lookup Rules

These are the rules which DTML uses to resolve names mentioned in `name=` and `expr=` tags. The rules are in order from first to last in the search path.

The DTML call signature is as follows:

```
def __call__(client=None, mapping={}, **kw)
```

The `client` argument is typically unreferenced in the body of DTML text, but typically resolves to the "context" in which the method was called (for example, in the simplest case, its client is the folder in which it lives).

The `mapping` argument is typically referred to as `_` in the body of DTML text.

The keyword arguments (ie `**kw`) are referred to by their respective names in the body of DTML text.

1. The keyword arguments are searched.
2. The mapping object is searched.
3. Attributes of the client, including inherited and acquired attributes, are searched.
4. If DTML is used in a Zope DTML Method or Document object and the variable name is `document_id` or `document_title` , then the id or title of the document or method is used.
5. Attributes of the folder containing the DTML object (its container) are searched. Attributes include objects in the contents of the folder, properties of the folder, and other attributes defined by Zope, such as `ZopeTime` . Folder attributes include the attributes of folders containing the folder, with contained folders taking precedence over containing folders.
6. User-defined Web-request variables (ie. in the `REQUEST.other` namespace) are searched.
7. Form-defined Web-request variables (ie. in the `REQUEST.form` namespace) are searched.
8. Cookie-defined Web-request variables (ie. in the `REQUEST.cookies` namespace) are searched.
9. CGI-defined Web-request variables (ie. in the `REQUEST.environ` namespace) are searched.