This is the official home for all Zope documentation.
The article explains the new high-level features and changes found in Zope.
You can have a look at the detailed change log to learn about all minor new features and bugs being solved in the latest release.

**Attention:** If you are upgrading from Zope 2, make sure you study the *From Zope 2 to Zope 4 or 5* documentation

- **What's new in Zope 5**
  - Dropped support for Python 2
  - Dropped support for **zServer**

- **What's new in Zope 4**
  - Restored sane version numbering
  - Extended Python version support
  - WSGI as the new default server type
  - View component Acquisition changes
  - Page Templates now rendered by Chameleon
  - Lower memory consumption at runtime
  - Simplified encoding configuration
  - Restyled Zope Management Interface (ZMI)
1.1 What’s new in Zope 5

1.1.1 Dropped support for Python 2

Zope 5 supports Python 3 only, versions 3.5 up through 3.8. All support code and special casing for Python 2, including the use of the six package, have been removed.

1.1.2 Dropped support for ZServer

As ZServer only supports Python 2, its integration has been removed. Only WSGI is now supported for web service.

1.2 What’s new in Zope 4

1.2.1 Restored sane version numbering

Version numbers for Zope have been confusing in the past. The original Zope project iterated through version one to two up to version 2.13. In parallel a separate project was launched using the name Zope 3. Zope 3 wasn’t a new version of the original Zope project and in hindsight should have used a different project name. These days this effort is known as BlueBream.

In order to avoid confusion between the separate Zope 3 project and a new version of this project, it was decided to skip ahead and use Zope 4.0 as the next version number. The increase in the major part of the version also indicates the number of backwards incompatible changes found in this release.

1.2.2 Extended Python version support

Zope 4 supports Python 2.7 and Python 3 versions 3.5 up to 3.8.

The Python 3 support currently covers the core dependencies shipped with Zope and is limited to the new WSGI based publisher.

Migrating an existing ZODB to Python 3 is not an automated process. You have to update to Zope 4 first, see From Zope 2 to Zope 4 or 5.

1.2.3 WSGI as the new default server type

Zope 2.13 first gained support for running Zope as a WSGI application, using any WSGI capable web server instead of the built-in ZServer.

Zope 4.0 takes this one step further and uses WSGI as the default setup. Functional testing (testbrowser) support also uses the new WSGI publisher.

The ZServer based publisher got moved into its own optional project. So if you rely on ZServer features, like Webdav, FTP, zdaemon or zopectl support, please make sure to install ZServer and use its mkzopeinstance script to create a Zope instance.

The ZServer project also includes limited functional testing support in the ZServer.Testing sub-package. testbrowser support is exclusively available based on the WSGI publisher, as a result of a switch from the unmaintained mechanize project to WebTest.
By default Zope only ships with a new `mkwsgiinstance` script which creates a Zope instance configured to run as a WSGI application. The example configuration uses the `waitress` web server, but Zope can be run using any WSGI capable web server.

To make running Zope easier, a new `runwsgi` command line script got added, which can read a PasteDeploy configuration and create and run the WSGI pipeline specified in it. By default such a configuration is created in the `etc/zope.ini` file. The `runwsgi` script supports both `-v/--verbose` and `-d/--debug` arguments to print out more information on the console. The debug argument enables Zope’s debug mode and disables the catch-all part of the `httpexceptions` WSGI middleware. This means unexpected and uncaught exceptions show their full traceback on the console and make it easier to debug them. Without debug mode, these exceptions result in a 500 Internal Server Error rendered as a normal HTML response.

**Note:** In case you experience an HTTP 500: Internal Server Error, where you would expect a Redirect, the following might help you.

Although the `zope.ini` created by `mkwsgiinstance` includes `egg:Zope#httpexceptions` as part of the pipeline, this might not be sufficient for existing projects. In case your project has configured a middleware handling and creating error views for HTTP exceptions, you need to make sure that `egg:Zope#httpexceptions` runs before that middleware. Otherwise a Redirect might not be handled as such. This can result in a non-functional ZMI.

The WSGI support has no built-in support for running as a daemon. Your chosen WSGI server might support this or you can use external projects like supervisor or systemd.

The WSGI support in Zope 4 has changed in a number of ways to make it more similar to its ZServer equivalent. In Zope 2.13 the WSGI support required using repoze WSGI middlewares to add transaction and retry handling. The WSGI support in Zope 4 no longer supports those middlewares but integrates transaction and retry handling back into the publisher code. This allows it to also add back full support for publication events and exception views. It does mean that the transaction is begun and committed or aborted inside the publisher code and you can no longer write WSGI middlewares that take part in the transaction cycle, but instead have to use Zope specific hooks like you do in the ZServer based publisher.

### 1.2.4 View component Acquisition changes

In Zope 2.12 Zope Toolkit view components changed and stopped inheriting from Acquisition base classes, as Acquisition got aware of `__parent__` pointers, which meant that `aq_parent(view)` worked, without the view having to mix-in an Acquisition base class. For backwards compatibility a new `AcquisitionBBB` class was mixed in, to continue to support calling `view.aq_parent`. This backwards compatibility class has been removed in Zope 4, so `view.aq_parent` no longer works and you have to use `aq_parent(view)`. The same applies for other view components like view page template files or viewlets.

### 1.2.5 Page Templates now rendered by Chameleon

Chameleon is an alternative implementation of the page template language supporting additional features and impressive template rendering speed.

So far it was available via the `five.pt` project. In Zope 4 the code from `five.pt` has been merged into Zope core and the Chameleon based engine is now the default, removing the need to install `five.pt` manually.

**Note:** The page template language parser in Chameleon is extremely strict. For example, in Zope 2, the parser does not care about opening and closing tags that are not matched in terms of being uppercase/lowercase, or unmatched.
opening/closing tags in general. All this will now cause template compilation to fail. See Page Template parsing issues for help.

1.2.6 Lower memory consumption at runtime

Zope 4 depends on a new DateTime release. The new release has been optimized for better memory use. Applications using a lot of DateTime values like the Plone CMS have seen total memory usage to decrease by 10% to 20% for medium to large deployments.

1.2.7 Simplified encoding configuration

As it is reasonable to have one unified encoding in ZMI and frontend, support for management_page_charset (as property of a folder) has been removed. default-zpublisher-encoding in zope.conf is the only place where to define the site encoding that governs how the ZPublisher and Zope Page Templates handle encoding and decoding of text.

1.2.8 Restyled Zope Management Interface (ZMI)

The ZMI (Zope Management Interface) is now styled with Bootstrap. See Migrating to the new bootstrap-based ZMI for details how to adapt Zope add-on packages to the new styling.
1.2. What's new in Zope 4

![ZMI root screenshot](localhost:8080/manage_main)

**Fig. 1:** The newly styled ZMI root
Fig. 2: The Ace editor on a page template. The editor is also used for Python Scripts, DTML Methods/Documents and Z SQL Methods.
1.2. What’s new in Zope 4

This application’s transactional feature allows you to easily revert the application’s settings or data. You can revert the application to any previous point in time.

Fig. 3: The central *Undo* view is reached from the new left-side menu
Properties allow you to assign simple values to Zope objects. To change properties, click the checkbox next to the property name and click Save Changes.

To add a new property, enter a name, type and value for the new property and click Save Changes.
This document describes installing Zope with `zc.buildout` (the **recommended** method) or via `pip`.

- **Prerequisites**

- **Installing Zope with `zc.buildout`**
  - Built-in standard buildout configuration
  - Custom buildout configurations
    - Minimum configuration
    - Using `plone.recipe.zope2instance`

- **Installing Zope with `pip`**

- **Building the documentation with Sphinx**

## 2.1 Prerequisites

In order to install Zope, you must have the following prerequisites available:

- A supported version of Python, including the development support if installed from system-level packages. Supported versions include **3.5** up to **3.8**.

- Zope needs the Python `zlib` module to be importable. If you are building your own Python from source, please be sure that you have the headers installed which correspond to your system’s `zlib`.

- A C compiler capable of building extension modules for your Python (gcc recommended).

- If you are using a Python interpreter shipping with your Linux distribution, you need to install the matching Python development package. As example, for Python 3 on Ubuntu 18.04, you have to type the following:
2.2 Installing Zope with zc.buildout

zc.buildout is a powerful tool for creating repeatable builds of a given software configuration and environment. The Zope developers use zc.buildout to develop Zope itself, as well as the underlying packages it uses. This is the recommended way of installing Zope.

Installing the Zope software using zc.buildout involves the following steps:

- Download and uncompress the Zope source distribution from PyPI if you are using the built-in standard buildout configuration
- Create a virtual environment
- Install zc.buildout into the virtual environment
- Run the buildout

The following examples are from Linux and use Zope version 4.1. Just replace that version number with your desired version.

2.2.1 Built-in standard buildout configuration

```bash
$ wget https://pypi.org/packages/source/Z/Zope/Zope-4.1.tar.gz
$ tar xfvz Zope-4.1.tar.gz
$ cd Zope-4.1
$ python3.7 -m venv .
$ bin/pip install -U pip zc.buildout
$ bin/buildout
```

2.2.2 Custom buildout configurations

Instead of using the buildout configuration shipping with Zope itself, you can also start with your own buildout configuration file.

The installation with a custom buildout configuration does not require you to download Zope first:

```bash
$ python3.7 -m venv zope
$ cd zope
<create buildout.cfg in this folder>
$ bin/pip install -U pip zc.buildout
$ bin/buildout
```

Minimum configuration

Here's a minimum buildout.cfg configuration example:

```
[buildout]
extends = https://zopefoundation.github.io/Zope/releases/4.1/versions-prod.cfg
parts =
```

(continues on next page)
Using `plone.recipe.zope2instance`

To make your life a lot easier, you can use `plone.recipe.zope2instance` to automate a lot of the configuration tasks from the following document, *Configuring and Running Zope*. `plone.recipe.zope2instance` has a myriad configuration options, please see the PyPI page.

```lfm
[buildout]
extends = https://zopefoundation.github.io/Zope/releases/4.1/versions-prod.cfg
parts =
  zopeinstance

[zopeinstance]
recipe = plone.recipe.zope2instance
user = admin:adminpassword
http-address = 8080
zodb-temporary-storage = off

[wsgi = /path/to/zope.ini]
```

One feature this kind of installation offers is the easy integration of WSGI servers other than the built-in `waitress`. You can specify a file path to a WSGI configuration file to use when starting the Zope instance. This works for WSGI servers that offer a PasteDeploy-compatible entry point, like `gunicorn`. You will need to create the `.ini` file yourself, and don’t forget to include the WSGI server software egg in the `eggs` specification:

```lfm
[zopeinstance]
recipe = plone.recipe.zope2instance
eggs =
  gunicorn
user = admin:adminpassword
http-address = 8080
zodb-temporary-storage = off
wsgi = /path/to/zope.ini
```

### 2.3 Installing Zope with `pip`

Installing the Zope software using `pip` involves the following steps:

- Create a virtual environment (There is no need to activate it.)
- Install Zope and its dependencies

Example steps on Linux. Replace the version number “4.1” with the latest version you find on https://zopefoundation.github.io/Zope/:
You can also install Zope using a single requirements file. **Note that this installation method might install packages that are not actually needed** (i.e. more than are listed in the `install_requires` section of `setup.py`):

```bash
$ bin/pip install -r https://zopefoundation.github.io/Zope/releases/4.1.3/requirements-full.txt
```

### 2.4 Building the documentation with Sphinx

If you have used `zc.buildout` for installation, you can build the HTML documentation locally:

```bash
$ bin/make-docs
```
 CHAPTER 3

Configuring and Running Zope

Whichever method you used to install Zope and create a server instance (see *Installing Zope*), the end result is configured and operated the same way.

**Note:** If you have installed Zope using `zc.buildout in conjunction with plone.recipe.zope2instance` as outlined in *Installing Zope*, many of the following tasks are already done for you and some others differ slightly. You can immediately skip down to *Running Zope*.

- *Creating a Zope instance*
- *Filesystem Permissions*
- *Configuring Zope*
  - `zope.ini`: WSGI configuration
  - `zope.conf`: Zope settings
- *Running Zope*
  - Running Zope in the foreground
  - Running Zope as a Daemon
  - Debugging Zope
  - Running scripts
  - Adding users
- *Running Zope (plone.recipe.zope2instance install)*
  - Running Zope in the foreground
  - Running Zope as a Daemon
  - Debugging Zope
3.1 Creating a Zope instance

Once you’ve installed Zope, you will need to create an “instance home”. This is a directory that contains configuration and data for a Zope server process. The instance home is created using the `mkwsgiinstance` script:

```bash
$ bin/mkwsgiinstance -d .
```

The `-d .` argument specifies the directory to create the instance home in.

You will be asked to provide a user name and password for an administrator’s account during `mkwsgiinstance`. To see all available command-line options, run the script with the `--help` option:

```bash
$ bin/mkwsgiinstance --help
```

If you followed the example and chose the current directory, you’ll find the instances files in the subdirectories of the `virtualenv`:

- `etc/` will hold the configuration files.
- `var/` will hold the database files.
- `var/log` will hold log files.
3.2 Filesystem Permissions

You need to set permissions on the directory Zope uses to store its data. This will normally be the var directory in the instance home. Zope needs to read and write data to this directory. Before running Zope you should ensure that you give adequate permissions to this directory for the user id Zope will run under.

Do not run Zope as root. Either create a user specifically for Zope or use an existing account with non-admin privileges.

3.3 Configuring Zope

Your instance’s configuration is defined in its etc/zope.conf and etc/zope.ini configuration files.

Note: Any changes in these configuration files require you to restart your Zope instance before they become active.

3.3.1 zope.ini: WSGI configuration

The file etc/zope.ini contains all settings related to the WSGI pipeline, the WSGI server and logging.

When starting Zope, if you see errors indicating that an address is in use, then you may have to change the ports Zope uses for HTTP. The default HTTP port used by Zope is 8080. You can change the port used by editing the [server] configuration section, which defines settings for the WSGI server itself:

```
[server:main]
use = egg:waitress#main
host = 127.0.0.1
port = 8080
```

See the section Using alternative WSGI server software to learn how to integrate WSGI servers other than the default waitress.

Zope is configured in the [app] section. It defines the so-called “entry point” called by the WSGI server and where the Zope configuration file zope.conf is located:

```
[app:zope]
use = egg:Zope#main
zope_conf = /path/to/zope.conf
```

The logging configurations are part of the etc/zope.ini file as well. The default configurations created by mkwsgiinstance and plone.recipe.zope2instance are suitable for most applications. The Python Logging Cookbook has a great selection of topics for advanced configurations.

Keep in mind that different WSGI servers have different logging behaviors. Some have their own access and event logging, some don’t log anything at all. For good control over your application’s logging needs, the default configurations use the translogger WSGI middleware from the Paste package. It can capture and log all errors propagating from your application.

Note: If your application is created using a custom zc.buildout configuration and you want to use translogger for logging, make sure to specify the wsgi extra for Zope in your buildout’s eggs specification, like Zope[wsgi].
3.3.2 zope.conf: Zope settings

You configure Zope itself in etc/zope.conf.

For a full description of the supported sections and directives for zope.conf, refer to the configuration reference section.

3.4 Running Zope

3.4.1 Running Zope in the foreground

To run Zope without detaching from the console, use:

```
$ bin/runwsgi -v etc/zope.ini
Starting server in PID 24934.
serving on http://127.0.0.1:8080
```

In this mode, Zope emits its log messages to the console, and does not detach from the terminal.

By default this command does not enable Zope’s debug mode, so it can be used for production.

In order to enable debug mode, you can add the additional -d or --debug argument to the command:

```
$ bin/runwsgi -dv etc/zope.ini
Starting server in PID 55111.
serving on http://127.0.0.1:8080
```

The runwsgi commands takes a PasteDeploy configuration file as its argument. You can configure different WSGI capable servers, the WSGI pipeline or logging configuration in this file.

Now you are able to log in using a browser, as described in Logging In To Zope.

3.4.2 Running Zope as a Daemon

Zope itself has no built-in support for running as a daemon any more.

If you create your Zope instance using plone.recipe.zope2instance you can use its start/stop script to dae-
monize Zope. See the next section for how to do that.

Alternatively, you can use projects like supervisord to achieve this or use your operating system’s built-in process manager, like systemd on most Linux versions. As an example, the following systemd service configuration works with the runwsgi script. It assumes your buildout is located at /opt/zopeinstance and the user account your Zope instance runs under is zope:

```
[Unit]
Description=Zope client zopeinstance
After=network.target

[Service]
Type=simple
User=zope
ExecStart=/opt/zopeinstance/bin/runwsgi /opt/zopeinstance/etc/zope.ini
KillMode=control-group
TimeoutStartSec=10
TimeoutStopSec=10
```

(continues on next page)
Save this configuration under /etc/systemd/system/zopeinstance.service and execute systemctl daemon-reload for systemd to read it. After that you can use standard systemctl commands to start, restart or stop the Zope instance:

```
[root@server]# systemctl start zopeinstance
[root@server]# systemctl restart zopeinstance
[root@server]# systemctl status zopeinstance
[root@server]# systemctl stop zopeinstance
```

### 3.4.3 Debugging Zope

In order to debug the Zope application, it can be helpful to connect to its database and inspect or change it on the command line. This feature was previously available via the dedicated zopectl debug command - in the new WSGI setup this is available via the zconsole module and console script:

```
$ bin/zconsole debug etc/zope.conf
>>> app
<Application at >

>>> app.acl_users
<UserFolder at /acl_users>

>>> import transaction
>>> transaction.begin()
>>> app.acl_users._doAddUser('foo', 'bar', ['Manager'], [])
<User 'foo'>
>>> transaction.commit()
```

### 3.4.4 Running scripts

This was previously available using zopectl run <path_to_script> <scriptarg1> ... Again in the WSGI setup the zconsole module and console script can be used:

```
$ bin/zconsole run etc/zope.conf <path_to_script> <scriptarg1> ...
```

### 3.4.5 Adding users

If you need to add a Manager to an existing Zope instance, you can do this using addzopeuser as follows:

```
$ bin/addzopeuser user password
```

The script expects to find the configuration file at etc/zope.conf.

3.4. Running Zope
3.5 Running Zope (plone.recipe.zope2instance install)

Script names and invocations vary slightly in installations that use `plone.recipe.zope2instance`, but the outcome is the same as described above. The following examples assume that the name of the buildout section was `zopeinstance`.

### 3.5.1 Running Zope in the foreground

To run Zope without detaching from the console, use:

```bash
$ bin/zopeinstance fg
...
Serving on http://127.0.0.1:8080
```

### 3.5.2 Running Zope as a Daemon

The `zopeinstance` runner script can daemonize the Zope process:

```bash
$ bin/zopeinstance start
...
daemon process started, pid=60116
```

Here’s how to get status information and how to stop the Zope instance:

```bash
$ bin/zopeinstance status
program running; pid=60116
$ bin/zopeinstance stop
...
daemon process stopped
```

To have your instance start automatically upon reboot, you will need to integrate with your operating system’s service startup facility. As an example, the following `systemd` service configuration works with the start/stop script generated by `plone.recipe.zope2instance`. It assumes the script name is `zopeinstance`, your buildout is located at `/opt/zopeinstance` and the user account your Zope instance runs under is `zope`:

```
[Unit]
Description=Zope client zopeinstance
After=network.target

[Service]
Type=forking
User=zope
ExecStart=/opt/zopeinstance/bin/zopeinstance start
PIDFile=/opt/zopeinstance/var/zopeinstance/Z4.pid
ExecStop=/opt/zopeinstance/bin/zopeinstance stop
ExecReload=/opt/zopeinstance/bin/zopeinstance stop && /opt/zopeinstance/bin/zopeinstance start
KillMode=control-group
TimeoutStartSec=10
TimeoutStopSec=10

[Install]
WantedBy=multi-user.target
```
Save this configuration under /etc/systemd/system/zopeinstance.service and execute `systemctl daemon-reload` for `systemd` to read it. After that you can use standard `systemctl` commands to start, restart or stop the Zope instance:

```
[root@server]# systemctl start zopeinstance
[root@server]# systemctl restart zopeinstance
[root@server]# systemctl status zopeinstance
[root@server]# systemctl stop zopeinstance
...
```

### 3.5.3 Debugging Zope

Debugging can be done at the command line:

```
$ bin/zopeinstance debug
Starting debugger (the name "app" is bound to the top-level Zope object)
>>> app
<Application at ...>
>>> app.acl_users
<OFS.userfolder.UserFolder object at ...>
>>> import transaction
>>> transaction.begin()
>>> app.acl_users._doAddUser('foo', 'bar', ['Manager'], [])
<User 'foo'>
>>> transaction.commit()
```

### 3.5.4 Running scripts

You can run Python scripts from the command line. The name `app` is injected into the top level namespace, it represents the root application object for your site.

```
$ bin/zopeinstance run <path_to_script> <scriptarg1> ...
```

### 3.5.5 Adding users

If you need to add a Manager to an existing Zope instance:

```
$ bin/zopeinstance adduser user password
Created user: user
```

### 3.6 Logging In To Zope

Once you’ve started Zope, you can then connect to the Zope webserver by directing your browser to:

```
http://yourhost:8080/manage
```

where ‘yourhost’ is the DNS name or IP address of the machine running Zope. If you changed the HTTP port as described, use the port you configured.
You will be prompted for a user name and password. Use the user name and password you provided in response to the prompts issued during the Zope instance creation, or configured into your buildout configuration for installs based on `plone.recipe.zope2instance`.

Now you’re off and running! You should be looking at the Zope management screen which is divided into two frames. On the left you can navigate between Zope objects and on the right you can edit them by selecting different management functions with the tabs at the top of the frame.

To create content to be rendered at `http://yourhost:8080/` create a Page Template or DTML Document named `index_html`.

### 3.7 Special access user accounts

#### 3.7.1 The Initial User

An initial username and password is needed to “bootstrap” the creation of normal managers of your Zope site. This is accomplished through the use of the ‘inituser’ file in the directory specified as the instance home.

The first time Zope starts, it will detect that no users have been defined in the root user folder. It will search for the ‘inituser’ file and, if it exists, will add the user defined in the file to the root user folder.

Normally, ‘inituser’ is created by the `makewsgiinstance` install script.

#### 3.7.2 The super user (“break glass in emergency” user)

If you find yourself locked out of your Zope instance you can create a user by placing a file named `access` in the directory specified as the instance home. The file has one line with a colon-separated login and password, like:

```
superuser:mysecretpassword
```

Now restart Zope and use these credentials to log in. This type of user account cannot create any content, but it can add new users to the user folder or edit existing users to get you out of a bind.

Do not forget to delete the `access` file and restart Zope when you are done.

### 3.8 Troubleshooting

- This version of Zope requires Python 3.5 and later. It will not run with any version of PyPy.
- To build Python extensions you need to have Python configuration information available. If your Python comes from an RPM you may need the python-devel (or python-dev) package installed too. If you built Python from source all the configuration information should already be available.
- See the `Change log` for important notes on this version of Zope.

### 3.9 Using alternative WSGI server software

The WSGI integration gives you a choice of WSGI server software to run your Zope application. This section lists several options that were selected because they either have a PasteDeploy entry point or have one provided by shim software, which means they work with the default Zope scripts for starting/stopping the service.
3.9.1 Things to watch out for

The ZODB uses connection pooling where a working thread grabs a connection from the pool to serve content and then releases it when the work is done. The default size of this connection pool is 7. You should choose a number of application threads that stays safely below that number of ZODB connections. If the WSGI server lets you configure the number of threads, 4 is a safe choice.

Another recommendation from Zope 2 is still valid as well: If you have a choice between less Zope instances with a higher number of threads each, or more instances with less threads each, choose the latter. Create more separate Zope instances and set the WSGI server threads value to e.g. 2.

**Warning:** If the WSGI server software lets you configure a number of worker processes, like gunicorn does, do not configure more than a single worker. Otherwise you will see issues due to concurrent ZODB access by more than one process, which may corrupt your ZODB.

3.9.2 Test criteria for recommendations

A simple contrived load test was done with the following parameters:

- 100 concurrent clients accessing Zope
- 100 seconds run time
- the clients just fetch “/”
- standard Zope 4 instances, one with ZEO and one without
- Python 2.7.16 on macOS Mojave/10.14.4
- standard WSGI server configurations, the only changes are to number of threads and/or number of workers where available.

This load test uncovered several issues:

- **cheroot** (tested version: 6.5.5) was magnitudes slower than all others. Unlike the others, it did not max out CPU. It is unclear where the slowdown originates. Others reached 500-750 requests/second. cheroot only served 12 requests/second per configured thread.
- **gunicorn** (tested version: 19.9.0) showed very strange behavior against the non-ZEO Zope instance. It serves around 500 requests/second, but then hangs and serves no requests for several seconds, before picking up again.
- **gunicorn** (tested version: 19.9.0) does not like the ZEO instance at all. No matter what configuration in terms of threads or workers was chosen gunicorn just hung so badly that even CTRL-C would not kill it. Switching to an asynchronous type of worker (tested with gevent) did not make a difference.
- **werkzeug** (tested version: 0.15.2) does not let you specify the number of threads, you only tell it to use threads or not. In threaded mode it spawns too many threads and immedialy runs up against the ZODB connection pool limits, so with Zope only the unthreaded mode is suitable. Even in unthreaded mode, the service speed was inconsistent. Just like gunicorn it had intermittent hangs before recovering.
- **bjoern** (tested version: 3.0.0) is the clear speed winner with 740 requests/second against both the ZEO and non-ZEO Zope instance, even though it is single-threaded.
- **waitress** (tested version: 1.3.0) is the all-around best choice. It’s just 10-15% slower than bjoern, but both the built-in WSGI tools as well as plone.recipe.zope2instance use it as the default and make it very convenient to use.
3.9.3 Recommended WSGI servers

waitress (the default and recommended choice)

If you create a Zope instance using the mkwsgiinstance script described above or the plone.recipe.zope2instance buildout recipe, you will automatically get a waitress-based server. The default configurations set up for you will be sufficient for most applications. See the [waitress documentation](#) for additional information.

Here's a very simple configuration using plone.recipe.zope2instance:

```ini
[zopeinstance]
recipe = plone.recipe.zope2instance
eggs =
zodb-temporary-storage = off
user = admin:password
http-address = 8080
```

Note the empty eggs section, you cannot leave it out.

waitress has many options that you can add to the buildout section. A full list is part of the [waitress documentation](#).

bjoern (the fastest)

The bjoern WSGI server can be integrated using a shim package called dataflake.wsgi.bjoern. See the [Using this package](#) section for details on how to integrate bjoern using Zope's own runwsgi script and how to create a suitable WSGI configuration.

If you use plone.recipe.zope2instance, the following section will pull in the correct dependencies:

```ini
[zopeinstance]
recipe = plone.recipe.zope2instance
eggs =
dataflake.wsgi.bjoern
zodb-temporary-storage = off
user = admin:password
http-address = 8080
wsgi = ${buildout:directory}/etc/bjoern.ini
```

3.9.4 Problematic WSGI servers

werkzeug

werkzeug is a WSGI library that contains not just a WSGI server, but also a powerful debugger. It can easily integrate with Zope using a shim package called dataflake.wsgi.werkzeug. See the [Using this package](#) section for how to integrate werkzeug using Zope's own runwsgi script and how to create a suitable WSGI configuration.

If you use plone.recipe.zope2instance, the following section will pull in the correct dependencies, after you have created a WSGI configuration file:

```ini
[zopeinstance]
recipe = plone.recipe.zope2instance
eggs =
dataflake.wsgi.werkzeug
zodb-temporary-storage = off
user = admin:password
```
gunicorn

The gunicorn WSGI server has a built-in PasteDeploy entry point and integrates easily. The following example build-out configuration section will create a bin/runwsgi script that uses gunicorn.

```ini
[gunicorn]
recipe = zc.recipe.egg
eggs =
   Zope
gunicorn
scripts =
   runwsgi
```

You can use this script with a WSGI configuration file that you have to create yourself. Please see the gunicorn documentation, especially the Configuration File section on Configuration Overview, for Paster Application configuration information. A very simple server configuration looks like this:

```ini
[server:main]
use = egg:gunicorn#main
host = 192.168.0.1
port = 8080
proc_name = zope
```

You can then run the server using runwsgi:

```
$ bin/runwsgi etc/gunicorn.ini
2019-04-22 11:45:39 INFO [Zope:45][MainThread] Ready to handle requests
Starting server in PID 84983.
```

Note: gunicorn version 19.9.0 or less will print an ominous warning message on the console upon startup that seems to suggest their WSGI entry point is deprecated in favor of using their own built-in scripts. This is misleading. Future versions will not show this message.

If you use plone.recipe.zope2instance, you can make it use gunicorn by adding its egg to the buildout section and setting the WSGI configuration file path to the path of the configuration file you created yourself:

```ini
[zopeinstance]
recipe = plone.recipe.zope2instance
eggs =
   gunicorn
zodb-temporary-storage = off
user = admin:password
http-address = 8080
wsgi = ${buildout:directory}/etc/gunicorn.ini
```

cheroot

The cheroot WSGI server can be integrated using a shim package called dataflake.wsgi.cheroot. See the Using this package section for details on how to integrate cheroot using Zope’s own runwsgi script and how to create a suitable

3.9. Using alternative WSGI server software
WSGI configuration.

If you use `plone.recipe.zope2instance`, the following section will pull in the correct dependencies:

```
[zopeinstance]
recipe = plone.recipe.zope2instance
eggs =
  dataflake.wsgi.cheroot
zodb-temporary-storage = off
user = admin:password
http-address = 8080
wsgi = ${buildout:directory}/etc/cheroot.ini
```

### 3.10 Debugging Zope applications under WSGI

You can debug a WSGI-based Zope application by adding a statement to activate the debugger. In addition, you can take advantage of WSGI middleware or debugging facilities built into the chosen WSGI server.

When developing your application or debugging, which is the moment you want to use debugging tools, you can start your Zope instance in *exceptions debug mode*. This will disable all registered exception views including `standard_error_message` so that exceptions are not masked or hidden.

This is how you run Zope in exceptions debug mode using the built-in `runwsgi` script:

```
$ bin/runwsgi -e etc/zope.ini
```

If you built your environment using `plone.recipe.zope2instance` you will need to do a manual change to your Zope configuration file. Enable exceptions debug mode by adding the `debug-exceptions` on setting before starting your application. The example presumes the Zope instance was named `zopeinstance`, your Zope configuration file will be at `parts/zopeinstance/etc/zope.conf`.

```
bin/zopeinstance fg
```

With Zope set up to let WSGI handle exceptions, these are a few options for the WSGI pipeline:

If you use `waitress`, you can make it output exception tracebacks in the browser by configuring `expose_tracebacks`. The keyword works in both standard and `plone.recipe.zope2instance` configurations:

```
[server:main]
use = egg:waitress#main
host = 127.0.0.1
port = 8080
expose_tracebacks = True
... or ...

[server:main]
paste.server_factory = plone.recipe.zope2instance:main
use = egg:plone.recipe.zope2instance#main
listen = 0.0.0.0:8080
threads = 2
expose_tracebacks = True
```

`werkzeug` includes a full-featured debugging tool. See the `dataflake.wsgi.werkzeug` documentation for how to enable the debugger. Once you’re up and running, the `werkzeug` debugger documentation will show you how to use it.
3.11 Zope configuration reference

`zodb.storage*`

`<blobstorage>` (ZODB.config.BlobStorage)

(blob-dir (string)

Path name to the blob storage directory.

`zodb.storage*`

`<demostorage>` (ZODB.config.DemoStorage)

(name (string)

The storage name, used by the `getName()` and `sortKey()` methods.

`zodb.storage*`

`<filestorage>` (ZODB.config.FileStorage)

(path (existing-dirpath)

Path name to the main storage file. The names for supplemental files, including index and lock files, will be computed from this.

(blob-dir (existing-dirpath)

If supplied, the file storage will provide blob support and this is the name of a directory to hold blob data. The directory will be created if it doesn’t exist. If no value (or an empty value) is provided, then no blob support will be provided. (You can still use a BlobStorage to provide blob support.)

create (boolean)

Flag that indicates whether the storage should be truncated if it already exists.

read-only (boolean)

If true, only reads may be executed against the storage. Note that the “pack” operation is not considered a write operation and is still allowed on a read-only filestorage.

quota (byte-size)

Maximum allowed size of the storage file. Operations which would cause the size of the storage to exceed the quota will result in a `ZODB.FileStorage.FileStorageQuotaError` being raised.

packer (string)

The dotted name (dotted module name and object name) of a packer object. This is used to provide an alternative pack implementation.

pack-gc (boolean) (default: true)

If false, then no garbage collection will be performed when packing. This can make packing go much faster and can avoid problems when objects are referenced only from other databases.

pack-keep-old (boolean) (default: true)

If true, a copy of the database before packing is kept in a “.old” file.

`<fullstorage>` (ZODB.config.BDBFullStorage)

(envdir (string)

interval (time-interval) (default: 2m)

kbyte (integer) (default: 0)
min (integer) (default: 0)
logdir (string)
cachesize (byte-size) (default: 128MB)
frequency (time-interval) (default: 0)
packtime (time-interval) (default: 4h)
gcpack (integer) (default: 0)
read-only (boolean) (default: off)

<mappingstorage> (ZODB.config.MappingStorage)
  name (string) (default: Mapping Storage)
    The storage name, used by the getName() and sortKey() methods.

<minimalstorage> (ZODB.config.BDBMinimalStorage)
envdir (string)
  interval (time-interval) (default: 2m)
kbyte (integer) (default: 0)
min (integer) (default: 0)
logdir (string)
cachesize (byte-size) (default: 128MB)
frequency (time-interval) (default: 0)
packtime (time-interval) (default: 4h)
gcpack (integer) (default: 0)
read-only (boolean) (default: off)

<temporarystorage> (tempstorage.config.TemporaryStorage)
  A nonundoing storage which keeps data in RAM and which does not need to be packed unless cyclic
  references are kept.
  name (string) (default: Temporary Storage)

<zeoclient> (ZODB.config.ZEOClient)
  server (*) (socket-connection-address)
  blob-dir (string)
    Path name to the blob cache directory.
  shared-blob-dir (boolean) (default: no)
    Tells whether the cache is a shared writable directory and that the ZEO protocol should not transfer
    the file but only the filename when committing.
  blob-cache-size (byte-size)
    Maximum size of the ZEO blob cache, in bytes. If not set, then the cache size isn’t checked and the
    blob directory will grow without bound.
    This option is ignored if shared_blob_dir is true.
blob-cache-size-check (integer)

ZEO check size as percent of blob_cache_size. The ZEO cache size will be checked when this many bytes have been loaded into the cache. Defaults to 10% of the blob cache size. This option is ignored if shared_blob_dir is true.

storage (string) (default: 1)

The name of the storage that the client wants to use. If the ZEO server serves more than one storage, the client selects the storage it wants to use by name. The default name is ‘1’, which is also the default name for the ZEO server.

cache-size (byte-size) (default: 20MB)

The maximum size of the client cache, in bytes, KB or MB.

name (string)

The storage name. If unspecified, the address of the server will be used as the name.

client (string)

Enables persistent cache files. The string passed here is used to construct the cache filenames. If it is not specified, the client creates a temporary cache that will only be used by the current object.

var (string)

The directory where persistent cache files are stored. By default cache files, if they are persistent, are stored in the current directory.

min-disconnect-poll (integer) (default: 5)

The minimum delay in seconds between attempts to connect to the server, in seconds. Defaults to 5 seconds.

max-disconnect-poll (integer) (default: 300)

The maximum delay in seconds between attempts to connect to the server, in seconds. Defaults to 300 seconds.

wait (boolean) (default: on)

A boolean indicating whether the constructor should wait for the client to connect to the server and verify the cache before returning. The default is true.

read-only (boolean) (default: off)

A flag indicating whether this should be a read-only storage, defaulting to false (i.e. writing is allowed by default).

read-only-fallback (boolean) (default: off)

A flag indicating whether a read-only remote storage should be acceptable as a fall-back when no writable storages are available. Defaults to false. At most one of read_only and read_only_fallback should be true.

username (string)

The authentication user name of the server.

password (string)

The authentication password of the server.

realm (string)
The authentication realm of the server. Some authentication schemes use a realm to identify the logic set of user names that are accepted by this server.

**drop-cache-rather-verify (boolean)** (default: off)

A flag indicating whether the client cache should be dropped instead of an expensive verification.

**client-label (string)**

A label for the client in server logs

---

**zodb.storage**

**zodb.storage**

**zodb.database**

Zope ZODB databases must have a name, and they are required to be referenced via the “zodb_db” database type because it is the only kind of database definition that implements the required mount-point argument. There is another database sectiotype named “zodb”, but it cannot be used in the context of a proper Zope configuration (due to lack of a mount-point).

**<zodb> (ZODB.config.ZODBDatabase)**

**zodb.storage**

**cache-size (integer)** (default: 5000)

Target size, in number of objects, of each connection’s object cache.

**cache-size-bytes (byte-size)** (default: 0)

Target size, in total estimated size for objects, of each connection’s object cache. “0” means no limit.

**large-record-size (byte-size)** (default: 16MB)

When object records are saved that are larger than this, a warning is issued, suggesting that blobs should be used instead.

**pool-size (integer)** (default: 7)

The expected maximum number of simultaneously open connections. There is no hard limit (as many connections as are requested will be opened, until system resources are exhausted). Exceeding pool-size connections causes a warning message to be logged, and exceeding twice pool-size connections causes a critical message to be logged.

**pool-timeout (time-interval)**

The minimum interval that an unused (non-historical) connection should be kept.

**historical-pool-size (integer)** (default: 3)

The expected maximum total number of historical connections simultaneously open.

**historical-cache-size (integer)** (default: 1000)

Target size, in number of objects, of each historical connection’s object cache.

**historical-cache-size-bytes (byte-size)** (default: 0)

Target size, in total estimated size of objects, of each historical connection’s object cache.

**historical-timeout (time-interval)** (default: 5m)

The minimum interval that an unused historical connection should be kept.
database-name (string)

When multi-databases are in use, this is the name given to this database in the collection. The name must be unique across all databases in the collection. The collection must also be given a mapping from its databases’ names to their databases, but that cannot be specified in a ZODB config file. Applications using multi-databases typical supply a way to configure the mapping in their own config files, using the “databases” parameter of a DB constructor.

allow-implicit-cross-references (boolean)

If set to false, implicit cross references (the only kind currently possible) are disallowed.

<zodb_db> (Zope2.Startup.datatypes.ZopeDatabase)

We need to specialize the database configuration section for Zope only by including a (required) mount-point argument, which is a string. A Zope ZODB database can have multiple mount points, so this is a multikey.

zodb.storage*

cache-size (integer) (default: 5000)

Target size, in number of objects, of each connection’s object cache.

cache-size-bytes (byte-size) (default: 0)

Target size, in total estimated size for objects, of each connection’s object cache. “0” means no limit.

large-record-size (byte-size) (default: 16MB)

When object records are saved that are larger than this, a warning is issued, suggesting that blobs should be used instead.

pool-size (integer) (default: 7)

The expected maximum number of simultaneously open connections. There is no hard limit (as many connections as are requested will be opened, until system resources are exhausted). Exceeding pool-size connections causes a warning message to be logged, and exceeding twice pool-size connections causes a critical message to be logged.

pool-timeout (time-interval)

The minimum interval that an unused (non-historical) connection should be kept.

historical-pool-size (integer) (default: 3)

The expected maximum total number of historical connections simultaneously open.

historical-cache-size (integer) (default: 1000)

Target size, in number of objects, of each historical connection’s object cache.

historical-cache-size-bytes (byte-size) (default: 0)

Target size, in total estimated size of objects, of each historical connection’s object cache.

historical-timeout (time-interval) (default: 5m)

The minimum interval that an unused historical connection should be kept.

database-name (string)

When multi-databases are in use, this is the name given to this database in the collection. The name must be unique across all databases in the collection. The collection must also be given a mapping from its databases’ names to their databases, but that cannot be specified in a ZODB config file. Applications using multi-databases typical supply a way to configure the mapping in their own config files, using the “databases” parameter of a DB constructor.
allow-implicit-cross-references (boolean)

If set to false, implicit cross references (the only kind currently possible) are disallowed.

mount-point (*) (Zope2.Startup.datatypes.mount_point)

The mount point is a slash-separated path to a ‘Products.ZODBMountPoint.Mount.MountPoint’ instance in Zope. If such an instance exists, it can mount an object (the mounted object) into Zope. By default, the object will be mounted at the same path in Zope (i.e. ‘/foo/bar’ in the database will be mounted at ‘/foo/bar’ in Zope).

The object can be mounted at a different point using the ‘virtual_path:real_path’ syntax (e.g. ‘mount-point /foo/bar:/bar’ will mount the object at ‘/bar’ in the database to ‘/foo/bar’ in Zope). The name of the mount point (‘bar’) must be the same as the mounted object.

It is also possible to specify the root that should be used in the mounted database by using the syntax ‘virtual_path:~real_root:real_path’. The root defaults to ‘Application’ and should not normally be changed.

collection-class (Zope2.Startup.datatypes.importable_name)

Change the connection class a database uses on a per-database basis to support different connection policies. Use a Python dotted-path name to specify the connection class.

class-factory (Zope2.Startup.datatypes.importable_name) (default: Zope2.Startup.datatypes.simpleClassFactory)

Change the class factory function a database uses on a per-database basis to support different class factory policy. Use a Python dotted-path name to specify the class factory function.

container-class (string)

Change the container class a (mounted) database uses on a per-database basis to support a different container than a plain Folder. Use a Python dotted-path name to specify the container class.

zope.product.base+

Product-specific configuration stanzas.

Products may use the <product-config> section type, or may supply a component.xml which defines section types with their own schemas.

All sections for this multisection will be collected into the ‘product_config’ attribute of the configuration object.

Base type for product-specific configuration sections.

Specific products should implement configuration sections by defining section types that implement this abstract type and using their own schema component to define meaningful settings.

product-config (null)

Product-specific configuration, expressed as arbitrary name-value pairs.

+ (string)

<environment> (Zope2.Startup.datatypes.environment)

A section which allows you to define simple key-value pairs which will be used as environment variable settings during startup.

+ (string)

Use any key/value pair, e.g. ‘MY_PRODUCT_ENVV AR foo_bar’

zodb.database

zodb.storage
**Zope Documentation, Release 5.0**

**zope.product.base**

Base type for product-specific configuration sections.

Specific products should implement configuration sections by defining section types that implement this abstract type and using their own schema component to define meaningful settings.

**<environment *> (Zope2.Startup.datatypes.environment)**

+ (string)

Use any key/value pair, e.g. ‘MY_PRODUCT_ENVVAR foo_bar’

**instancehome (existing-directory)**

The top-level directory which contains the “instance” data for the application server. It may also contain “etc”, “bin”, “log”, and “var” directories depending on how you’ve configured your Zope instance.

**clienthome (existing-directory) (metadefault: $instancehome/var)**

The directory used to store the default filestorage file used to back the ZODB database, as well as other files used by the Zope applications server during runtime.

**debug-mode (boolean) (default: off) (metadefault: off)**

A switch which controls several aspects of Zope operation useful for developing under Zope. When debug mode is on:

- Errors in product initialization will cause startup to fail (instead of writing error messages to the event log file).
- Filesystem-based scripts such as skins, PageTemplateFiles, and DTMLFiles can be edited while the server is running and the server will detect these changes in real time. When this switch is off, you must restart the server to see the changes.

Setting this to ‘off’ when Zope is in a production environment is encouraged, as it speeds execution (sometimes dramatically).

**debug-exceptions (boolean) (default: off) (metadefault: off)**

This switch controls how exceptions are handled. If it is set to “off” (the default), Zope’s own exception handling is active. Exception views or a standard_error_message are used to handle them.

If set to “on”, exceptions are not handled by Zope and can propagate into the WSGI pipeline, where they may be handled by debugging middleware.

This setting should always be “off” in production. It is useful for developers and while debugging site issues.

**locale (locale) (metadefault: unset)**

Locale name to be used. See your operating system documentation for locale information specific to your system. If the requested locale is not supported by your system, an error will be raised and Zope will not start.

**datetime-format (Zope2.Startup.datatypes.datetime_format) (default: us) (metadefault: us)**

Set this variable either to “us” or “international” to force the DateTime module to parse date strings either with month-before-days-before-year (“us”) or days-before-month-before-year (“international”). The default behaviour of DateTime (when this setting is left unset) is to parse dates as US dates.

**python-check-interval (integer) (default: 1000)**

Value passed to Python’s sys.setcheckinterval() function. The higher this is, the less frequently the Python interpreter checks for keyboard interrupts. Setting this to higher values also reduces the frequency of potential thread switches, which can improve the performance of a busy server.
http-realm (string) (default: Zope) (metadefault: Zope)

The HTTP “Realm” header value sent by this Zope instance. This value often shows up in basic authentication dialogs.

automatically-quote-dtml-request-data (boolean) (default: on) (metadefault: on)

Set this directive to ‘off’ in order to disable the autoquoting of implicitly retrieved REQUEST data by DTML code which contains a ‘<’ when used in <dtml-var> construction. When this directive is ‘on’, all data implicitly retrieved from the REQUEST in DTML (as opposed to addressing REQUEST.somevarname directly) that contains a ‘<’ will be HTML-quoted when interpolated via a <dtml-var> or &dtml- construct. This mitigates the possibility that DTML programmers will leave their sites open to a “client-side trojan” attack.

zmi-bookmarkable-urls (boolean) (default: on) (metadefault: on)

Set this directive to ‘on’ to cause Zope to show the ZMI right hand frame’s URL in the browser navigation bar as opposed to the static ‘/manage’. The default is ‘on’. To restore the behavior of Zope 2 where the URL was always static unless you opened the right-hand frame in its own browser window, set this to off.

pid-filename (existing-dirpath) (metadefault: $clienthome/Z4.pid)

The full path to which the Zope process will write its OS process id at startup.

trusted-proxy (*) (ipaddr-or-hostname) (metadefault: unset)

Define one or more ‘trusted-proxies’ keys, each of which is a hostname or an IP address. The set of definitions comprises a list of front-end proxies that are trusted to supply an accurate X_FORWARDED_FOR header to Zope (security-related).

max-conflict-retries (integer) (default: 3)

The maximum number of retries on a conflict error.

security-policy-implementation (Zope2.Startup.datatypes.security_policy_implementation) (default: C) (metadefault: C)

The default Zope “security policy” implementation is written in C. Set this key to “PYTHON” to use the Python implementation (useful for debugging purposes); set it to “C” to use the C implementation.

skip-authentication-checking (boolean) (default: off) (metadefault: off)

Set this directive to ‘on’ to cause Zope to prevent Zope from attempting to authenticate users during normal operation. Potentially dangerous from a security perspective. Only works if security-policy-implementation is set to ‘C’.

skip-ownership-checking (boolean) (default: off) (metadefault: off)

Set this directive to ‘on’ to cause Zope to ignore ownership checking when attempting to execute “through the web” code. By default, this directive is off in order to prevent ‘trojan horse’ security problems whereby a user with less privilege can cause a user with more privilege to execute code which the less privileged user has written.

verbose-security (boolean) (default: off) (metadefault: off)

Set this directive to ‘on’ to enable verbose security exceptions. This can help you track down the reason for Unauthorized exceptions, but it is not suitable for public sites because it may reveal unnecessary information about the structure of your site. Only works if security-policy-implementation is set to ‘PYTHON’.

default-zpublisher-encoding (Zope2.Startup.datatypes.default_zpublisher_encoding) (default: utf-8)

This key controls what character set is used to encode unicode data that reaches ZPublisher without any other specified encoding.
enable-ms-public-header (boolean) (default: off) (metadefault: off)

Set this directive to ‘on’ to enable sending the “Public” header in response to an WebDAV OPTIONS request - but only those coming from Microsoft WebDAV clients.

Though recent WebDAV drafts mention this header, the original WebDAV RFC did not mention it as part of the standard. Very few web servers out there include this header in their replies, most notably IIS and Netscape Enterprise 3.6.

Documentation about this header is sparse. Some versions of Microsoft Web Folders after 2005 apparently require its presence, but most documentation links have expired.
These documents outline migration steps that may be required when upgrading Zope.

### 4.1 From Zope 2 to Zope 4 or 5

#### 4.1.1 Zope products that are now distributed separately

During the Zope 4 development, several packages that used to be included have been separated out and ship separately.

On this page the term “add to your application buildout” is used generically. How you do so depends on your chosen installation method, like adding an egg name to a requirements file for pip installs, or adding it to an eggs specification for zc.buildout.

**ZServer**

ZServer does not support Python 3. The following only applies for Zope 4 under Python 2.

If you want to continue using ZServer instead of moving to WSGI you need to add the ZServer egg to your application buildout.

If you use the recipe `plone.recipe.zope2instance` in a buildout, add it to its eggs attribute and also add the flag `wsgi = off`.

**Sessioning**

If you have used (or want to use) the built-in support for sessioning, add the egg `Products.Sessions` to your application buildout.

You also need to make sure that your Zope configuration file contains a ZODB configuration for a temporary folder like this:
External Methods

If you have External Method objects in your ZODB, make sure to add the egg `Products.ExternalMethod` to your application buildout.

Site Error Log

If you have used the Site Error Log (the `error_logs` objects that show information about errors occurring in your application), add the egg `Products.SiteErrorLog` to your application buildout.

4.1.2 Migrating Zope product code

The following list shows a few common migration issues independent of the Python version used.

- **Changed import paths**
- **implementer** versus **implements**
- **Migrating to the new bootstrap-based ZMI**
  - Update existing package code
  - Custom icons and resources
  - Use custom resources via ZMI
  - Update existing Zope 2 ZMI templates
  - More details

**Changed import paths**

Several commonly used Zope code modules have moved. Here’s a short list of corrections you will have to make in your code. Many of these changed paths have already existed under Zope 2, so you can make those corrections ahead of time.

```python
from Globals import InitializeClass  # OLD
from AccessControl.class_init import InitializeClass  # NEW
```

**implementer** versus **implements**

How to signal that a class implements a specific interface has moved from a function called at class level into a class decorator and changed its name:
from zope.interface import implementer
from zope.interface import implements
from .interfaces import IMyClass

class MyClass:
    implements(IMyClass)  # OLD
    ...

@implementer(IMyClass)  # NEW
class AnotherClass:
    ...

**Migrating to the new bootstrap-based ZMI**

Since Zope 4.0b6 the ZMI is styled using Bootstrap. The previously used GIF icons were replaced by font glyphs which are stored in the package `zmi.styles` (part of Zope) together with the CSS and JavaScript needed by Bootstrap. The free Font Awesome glyphs are used as icons, see the table of available icons.

**Update existing package code**

If you have a Product or package which contains types, which can be added via the ZMI, the default icon will be shown.

To use one of the new icons add an attribute named `zmi_icon` to the class. As value use a name listed on available icons prefixed by `fas fa-`. Example to use the info icon (i in a circle):

```python
zmi_icon = 'fas fa-info-circle'
```

A few Zope products provide content that can be added in the ZMI without showing a dialog to collect data such as an id or title. These will now default to showing the new modal dialog as well. You can prevent that by adding another class variable:

```python
zmi_show_add_dialog = False
```

**Custom icons and resources**

To use custom icons (which are not part of `zmi.styles`) or load custom CSS resp. JavaScript, you have to:

1. create a directory and fill it with your assets
2. register this directory as resource directory
3. register a subscription adapter for `App.interfaces.ICSSPaths` resp. `App.interfaces.IJSPaths`. This adapter has to return an iterable of paths resp. URLs which should be loaded when rendering the ZMI.

Example taken from `zmi.styles`:

- Register the resource directory via ZCML:

  ```xml
  <browser:resourceDirectory
    name="zmi"
directory="resources" />
  ```

- Create a subscription adapter returning the path to the CSS file (`zmi.styles` has this code in `subscriber.py`):
import zope.component
import zope.interface

@zope.component.adapter(zope.interface.Interface)
def css_paths(context):
    """Return paths to CSS files needed for the Zope 4 ZMI."""
    return ('/++resource++zmi/bootstrap-4.1.1/bootstrap.min.css',
             '/++resource++zmi/fontawesome-free-5.1.0/css/all.css',
             '/++resource++zmi/zmi_base.css',
          )

• Register the subscriber via ZCML:

<subscriber
    provides="App.interfaces.ICSSPaths"
    factory=".subscriber.css_paths" />

Use custom resources via ZMI

To add custom CSS or JavaScript resources via ZMI you have to add a property:

• zmi_additional_css_paths for additional CSS
• zmi_additional_js_paths for additional JavaScript

The properties can have one of the following types:

• string
• ustring
• ulines

The value of the property has to be one or more paths/URLs to CSS resp. JavaScript which will be included in the HTML of the ZMI. (Paths have to be resolvable by the browser aka not simple file system paths.)

Update existing Zope 2 ZMI templates

The old Zope 2 styling rules did not allow a modern and responsive design. Now the Zope 4 ZMI uses some basic CSS class names of the bootstrap CSS framework and structuring concepts for page layout and forms. A ZMI page usually sequences following templates nesting the page core:

1. manage_page_header()
2. manage_tabs()
3. page core
4. manage_page_footer()

The page core of any form or object listing ZMI template is starting by the html element <main class="container-fluid">. Usually <main> is nesting a p element for a help-text and the actual form. To make specific form styling possible the form-element has following CSS names:

1. zmi-$classname
2. zmi-edit|-add
In general specific functional ZMI elements are declared by a CSS class with a prefixed zmi- whereas the basic layout is done by usual bootstrap classes following the typical element nesting: any form element has a bootstrap-like form-group structure containing a label and an input field. Important: The width of the input field is defined by the nesting div container using the responsive grid classes col-sm-9 col-md-10. With the classes col-sm-3 col-md-2 for the label, a complete bootstrap row of 12 is filled.

```html
<div class="form-group row">
  <label for="title" class="form-label col-sm-3 col-md-2">Title</label>
  <div class="col-sm-9 col-md-10">
    <input id="title" class="form-control" type="text" name="title" value="<dtml-if --title>&dtml-title;</dtml-if" />
  </div>
</div>
```

The following buttons are constructed as div element with the classname zmi-controls; the buttons use systematically the bootstrap class pair btn btn-primary.

```html
<div class="zmi-controls">
  <input class="btn btn-primary" type="submit" name="submit" value="Save" />
</div>
```

The following example code shows a whole restructured DTML template rendering the Zope 4 ZMI:

**Example: updated DTML template (from: ../Zope/src/OFS/dtml/documentEdit.dtml)**

```html
<dtm-var manage_page_header>
<dtm-var manage_tabs>
<main class="container-fluid">
  <p class="form-help">
    You may edit the source for this document using the form below. You may also upload the source for this document from a local file. Click the <em>browse</em> button to select a local file to upload.
  </p>
  <form action="manage_edit" method="post" class="zmi-dtml zmi-edit">
    <div class="form-group row">
      <label for="title" class="form-label col-sm-3 col-md-2">Title</label>
      <div class="col-sm-9 col-md-10">
        <input id="title" class="form-control" type="text" name="title" value="<dtml-if title>&dtml-title;</dtml-if" />
      </div>
    </div>
    <div class="form-group">
      <textarea id="content" data-contenttype="html" class="form-control zmi-code col-sm-12" name="data:text" wrap="off" rows="20"><dtml-var __str__></textarea>
    </div>
  </form>
</main>
```

(continues on next page)
More details

**Textarea:** A text area element for editing template code or script code uses the JS library [ace](http://ace.sh) for syntax high-lighting and line numbering. **Textarea** elements which are declared by the CSS class `zmi-code` are transformed into an [ace](http://ace.sh) editor field. Moreover this element has an attribute `data-contenttype` which is needed by ace-editor to determine the fitting syntax highlighting. **ZPT-Example** see: ../Zope/src/Products/PageTemplates/www/ptEdit.zpt

**File upload element:** The file upload element has its own form container (classified as `zmi-upload`). All subsequent elements are nested as `input-group` containing a `div` classified as `custom-file` nesting the actual input element. An inline JS fired on the `onchange` event beautifies the file name shown after selecting it. **ZPT-Example** see: ../Zope/src/Products/PageTemplates/www/ptEdit.zpt

**Hints and Warnings:** Some input fields show additional information; these are added as element `<small>` directly following the referred input field. (Both elements are nested by the width defining `div` container). Possible text colors are declared by typical bootstrap class names like `text-warning`.

**Icons:** Zope 4 object classes which are shown in the ZMI have declared a class variable `zmi_icon`; this string
corresponds to an appropriate font icon-CSS class supplied by the Fontawesome web font (https://fortawesome.com/icons).

Tables: Bootstrap requires an explicit CSS class `table` for any table; especially long item lists should get an additional CSS class `table-sm` and maybe another class `table-striped` for a better readability. Finally it is recommended to add a specific id attribute like `zmi-db_info`. The general table structure is compliant to bootstrap standard table (https://getbootstrap.com/docs/4.1/content/tables/).

ZMI-classes: All basic styling of the zmi-elements is defined in the CSS file, see: `../Zope/src/zmi/styles/resources/zmi_base.css`

Implicit handling of old Zope 2 ZMI templates: Old templates which do not contain the `<main>` element are automatically post-processed by a JavaScript function in the browser. The DOM is minimally modified, so that old forms will fit `somehow` into the Zope 4 layout. In the page footer a hint about this automatically customizing is shown.

### 4.1.3 Migrating content

These issues may appear when rendering content (templates, scripts or other built-in Zope code objects) created with Zope 2 in Zope 4.

**Page Template parsing issues**

Zope 4 is using Chameleon as its new parsing engine for Page Templates. Chameleon is strict. **Very strict.** Have I mentioned that Chameleon’s parsing is **extremely strict**? It will throw any sloppy HTML and TAL/TALES right in your face and refuse to compile it, even if it may be syntactically correct and the Zope 2 parsing engine has worked with it just fine.

- namespace names are case-sensitive. For Page templates, that means only lowercased namespaces like `tal` or `metal` are allowed:

  ```html
  <span TAL:CONTENT="string:foo"></span> <!-- BAD namespace TAL -->
  <span Tal:content="string:foo"></span> <!-- BAD namespace Tal -->
  <span tal:content="string:foo"></span> <!-- OK -->
  ```

- Opening and closing tags must match in type:

  ```html
  <tr>...
  <td>...
  </td> <!-- BAD: Mismatched open/close tag -->
  ```

- Opening and closing tags must match in case:

  ```html
  <a href=".">Text</a> <!-- BAD: Mismatched open/close tag -->
  <a href=".">Text</a> <!-- OK -->
  <A href=".">Text</A> <!-- OK -->
  <td>... </td> <!-- BAD: Mismatched open/close tag -->
  <td>... </td> <!-- OK -->
  ```

- HTML comments must not contain any double hyphens inside the comment or more than two hyphens in the closing sequence:

  ```html
  <!-- OK -->
  <!-- BAD: Mismatched open/close tag -->
  <!-- BAD: Mismatched open/close tag -->
  ```
• HTML syntax errors that were ignored before

```html
<a href="." class="mystyle">...</a> <!-- BAD: extraneous " -->
```

• Python expression syntax errors that were parsing OK under Zope 2 but caused errors at execution time are now caught during parsing:

```html
<a href="">
    tal:attributes="href python:context.myfunc(a=1, a=1)"
</a>
<!-- BAD: Python syntax error
```

To help identify such issues, code for a *External Method* that searches the ZODB for Page Templates and reveals errors has been provided. Make sure you have the `Products.ExternalMethod` egg in your application configuration before following these steps:

- In the ZMI root, instantiate a *External Method* object from the dropdown at the top right.
- Give it an id of your choosing and an optional title. For *Module Name* enter `Products.PageTemplates.find_bad_templates`, and for *Function Name* `find_bad_templates`. Click on *Save Changes*.
- Visit the *Test* tab at the top. The process time will vary with the size of your ZODB object tree and the number of Page Templates found. The report page will identify each Page Template that does not compile cleanly and point out the issues.

Run the script after each round of fixes as the parser will stop after the first error it encounters, even if there are more errors in a template.

### 4.1.4 Migrating the ZODB

This document describes the process of migrating a ZODB created with Zope 2 into a Zope 4 environment. The migration example steps have been tested on a `FileStorage`-based ZODB with a `Data.fs` file.

**Warning:** As soon as you open a ZODB from Zope 2 under Zope 4 you cannot use it under Zope 2 anymore, regardless of how the ZODB is opened (direct access to a `Data.fs` file or indirect access through a ZEO server). Always work on a copy of your ZODB so you retain a working copy for Zope 2 if you need to go back.

- **Pre-migration steps on Zope 2**
  - Prepare ZODB-based code
  - Delete ZODB objects that no longer exist under Zope 4
- **Migrate to Zope 4 on Python 2**
- **Migrate to Zope 4 on Python 3**
  - Migration example
    * Going from Zope 2 to Zope 4
    * Going from Python 2 to Python 3
  - Finding broken scripts and templates
  - If you encounter `UnicodeDecodeError` exceptions
- **Under the hood: Changes in ZODB storage on Python 3**
Pre-migration steps on Zope 2

The following pre-migration steps can be done while still on Zope 2 and will ease the final process.

Prepare ZODB-based code

Syntax changes that come with the move from Python 2 to Python 3 for filesystem code apply to ZODB code as well, such as Python Scripts, DTML Methods, DTML Documents, Z SQL Methods and Page Templates. Typical issues include:

• switching `print` statements to `print` function call syntax
• switching removed `string` module function calls to their string method equivalents
• safe handling of changed return value types for dictionary methods, such as `keys`, `values` or `items`
• fix indentation where a mix of spaces and tabs is used
• etc.

Many of these and others will be familiar from changing filesystem code to be Python 3 compatible.

Delete ZODB objects that no longer exist under Zope 4

The Control_Panel has seen changes in Zope 4 that have a risk of introducing spurious errors when verifying the ZODB contents in the steps below. Visit the ZMI while still running on Zope 2 and delete all objects you see in the Products folder at `/Control_Panel/Products/manage_main`. Pack the ZODB after the cleanup.

Migrate to Zope 4 on Python 2

There are no specific ZODB-related migration steps to take when moving to a Python 2-based Zope 4 environment, except when you’re proceeding with a Python 3 migration. See the section Going from Zope 2 to Zope 4 below for details.

Migrate to Zope 4 on Python 3

This part describes the process of migrating a ZODB created with Python 2 (using Zope 2 or 4) to Python 3 (using Zope 4). As there are significant changes between the two platforms, there is no automated process to cover all edge cases, so it is necessary to prepare and test your migration well in advance.
Migration example

- **Back up your ZODB before proceeding**
- Make all ZODB-persisted code Python 3 compatible (see above), while keeping Python 2 compatibility.
- Test that converted code works as expected

**Going from Zope 2 to Zope 4**

If your ZODB was created under Zope 2 you have a few additional steps that will ensure the latest ZODB code under Python 3 will work with your ZODB data. Make sure your ZODB is packed before going on.

- prepare a Python 2 environment containing...
  - Zope 4 (latest)
  - all relevant applications and addons for your ZODB
  - zodbverify
- prepare a Zope configuration
  - Create a new Zope instance using `mkwsgiinstance` or `plone.recipe.zope2instance` build-out configuration
  - make sure the created configuration files (under `etc/` if you used `mkwsgiinstance` and under `parts/` `<INSTANCE_NAME>/etc` if you used `plone.recipe.zope2instance`) reflect what was in your Zope 2 configuration before the migration
  - start the Application using `bin/runwsgi etc/zope.ini` or `bin/<INSTANCE_NAME>`, depending on the mechanism you used to create the instance configuration. Test it intensively for incompatibilities and errors.
- shut down the Zope instance(s) and ZEO server that serves your ZODB
- `run bin/zodbverify -f path/to/Data.fs` to uncover any errors in your ZODB. You may see cryptic errors pointing to the `Products` attribute of the `Control_Panel`, this is not critical. All others need to be fixed.

Now you have a ZODB that is ready to be opened under Python 3 for the remaining steps.

**Going from Python 2 to Python 3**

- Prepare a Python 3 environment, containing:
  - Zope 4 (latest),
  - all relevant applications and addons for your ZODB, (make sure they are compatible with Python 3)
  - zodbupdate
  - zodbverify
- Prepare a Zope configuration
  - Create a new Zope instance using `mkwsgiinstance` or `plone.recipe.zope2instance` build-out configuration
  - make sure the created configuration files (under `etc/` if you used `mkwsgiinstance` and under `parts/` `<INSTANCE_NAME>/etc` if you used `plone.recipe.zope2instance`) reflect what was in your Zope 2 configuration before the migration
• make sure the Zope instance(s) and ZEO server that serves your ZODB are shut down
• to prevent any compatibility issues with the ZODB index files created under Python 2, remove Data.fs.
  index before proceeding.
• run the ZODB conversion. Please note that you cannot use -n to use the nondestructive --dry-run mode
  at this moment, but the actual conversion works: bin/zodbupdate --pack -f var/filestorage/
  Data.fs --convert-py3 --encoding utf-8 --encoding-fallback latin1
• Verify the ZODB by iterative loading every pickle using bin/zodbverify -f path/to/Data.fs
• Start the Application using bin/runwsgi etc/zope.ini or bin/<INSTANCE_NAME>, depending on
  the mechanism you used to create the instance configuration.
• Verify that the Application works as expected.
• If your application uses the ZCatalog and there are problems with any of them, do a clear and rebuild.

Finding broken scripts and templates

You can find most scripts and templates that no longer compile under Python 3 by visiting the ZMI edit tabs, where
you will see error messages for e.g. syntax errors. Page Templates that have Python expressions embedded can only
be diagnosed at run time with manual site testing.

The ZMI edit tab method can be scripted as well by emulating what happens behind the scenes. You can write a script
that uses e.g. ZopeFind to find objects of those script-like types and then calling the methods that attempt to compile
the script content, such as...

• pt_macros() for Page Templates, which will store errors in an attribute _v_errors that you can read out
• _compile() on Python Scripts that will store errors in an attribute errors that you can read out, or the call
  will directly raise a SyntaxError
• template.cook() for Z SQL Methods, which will raise an exception of type DocumentTemplate.
  DT_Util.ParseError if there are problems
• cook() for DTML Methods and DTML Documents, which will raise an exception of type
  DocumentTemplate.DT_Util.ParseError if there are problems

If you encounter UnicodeDecodeError exceptions

If zodbupdate or the Application raises a UnicodeDecodeError after startup, there are several things to con-
side:

If the error happens on an object of a Product that is not migrated yet, you can add an entry_point in setup.py
for the package containing the persistent Python classes. The entry point has to be named "zodbupdate.decode"
and needs to point to a dictionary mapping paths to str attributes to a conversion (binary resp. a specific encoding).
For details, see zodbupdate documentation and or a code example in PythonScripts.

Under the hood: Changes in ZODB storage on Python 3

This section provides deeper technical detail about how the move to Python 3 affects the ZODB.
The string problem

A ZODB `Data.fs` which was created under Python 2 cannot be opened under Python 3. This is prevented by using a different magic code in the first bytes of the file. This is done on purpose because `str` has a different meaning for the two Python versions: Under Python 2, a `str` is a container for characters with an arbitrary encoding (aka `bytes`). Python 3 knows `str` as a text datatype which was called `unicode` in Python 2.

Trying to load a `str` object in Python 3 which actually contains binary data will fail. It has to be bytes, but `bytes` is an alias for `str` in Python 2. This means Python 2 replaces `bytes` with `str`, making it impossible to give Python 3 the class it expects for binary data. A Python 2 `str` with any non-ascii characters will break, too.

For more details, read the Saltlab-Sprint notes from Harald Frisnegger

The string solution

The `Data.fs` has to be migrated: each `str` which actually contains `bytes` has to be converted into a `zodbpickle.binary` object which deserialises as `bytes` under Python 3. The `str` objects actually containing text have to be decoded to `str` (known as `unicode` in Python 2).

The code problem

Python 3 is not backwards-compatible to Python 2 in terms of its syntax, which is a problem for Persistent objects in the ZODB containing Python code. This is problem might arise with PythonScript objects, and TAL or DTML templates that contain Python statements or expressions.

The code solution

There are several tools that help with getting your code ready for Python 3, especially in large code bases:

- `2to3` comes with modern Python distributions preinstalled and can be used to convert either extracted code in files or directly on the ZODB through a custom script.

- `gocept.template_rewrite` can extract and rewrite code parts of template files (DTML, ZPT).

- `zodbsync` is a tool to serialize ZODB objects and store them in a file system tree and restore ZODB them from the same structure.

The migration path heavily depends on your specific use case and can range from manually finding, inspecting and fixing code objects to setting up a large, auditable and automated process. The tooling referenced above even allows users to extract code to a file system, convert it and restoring it back to the ZODB while keeping changes under version control.

Further reading

The Plone project documentation contains a section Migrate a ZODB from Python 2.7 to Python 3
CHAPTER 5

Maintainer information

Note: This is internal documentation, mostly for Zope maintainers who manage software releases and the documentation

Contents

• Maintainer information
  – Release process
    • Maintainers
    • Steps for creating a new Zope release
  – Maintaining the Zope documentation
    • Contributing to the documentation
    • Building the documentation
    • The official documentation site on Read the Docs

5.1 Release process

5.1.1 Maintainers

The following persons have access to the Zope package on PyPI (in order to release new versions):

• Hanno Schlichting
• Michael Howitz
• Tres Seaver
5.1.2 Steps for creating a new Zope release

- Create releases for the packages mentioned in buildout.cfg below auto-checkout enter them into versions-prod.cfg and run bin/buildout to update requirements-full.txt.
- Garden the change log and check it for spelling issues.
- Check the future PyPI long description for ReST errors:
  ```
  $ bin/longtest
  ```
- Check in the changes.
- Update version information in change log and setup.py:
  ```
  $ bin/prerelease
  ```
- Pin the Zope version in versions-prod.cfg.
- Run bin/buildout to update requirements-full.txt.
- Commit the changes.
- Run all tests:
  ```
  $ bin/tox
  ```
- Upload the tagged release to PyPI:
  ```
  $ bin/release
  
or
  $ git tag -as <TAG-NAME> -m "- tagging release <TAG-NAME>"
  $ git push --tags
  $ bin/zopepy setup.py egg_info -Db '' sdist bdist_wheel
  $ bin/twine upload -s dist/Zope-<TAG-NAME>*
  ```
- Update version information:
  ```
  $ bin/postrelease
  $ vi versions-prod.cfg (remove Zope pin)
  $ bin/buildout
  ```
- Commit and push the changes.
- Check that the changes have been propagated to https://zope.readthedocs.io/en/latest/changes.html. (This should be done automatically via web hooks defined in GitHub and RTD.)
- Update https://zopefoundation.github.io/Zope/:
  ```
  $ git checkout gh-pages
  $ python3.7 build_index.py
  ```
- Add the newly created files and commit and push the changes.
• Check the versions.cfg file for outdated or updated packages and update version information where necessary. You can get a good overview of outdated packages on requires.io:

```
$ bin/checkversions versions-prod.cfg
$ bin/checkversions versions.cfg
$ bin/buildout
```

**Note:** This step is done after the release to have time to fix problems which might get introduced by new versions of the dependencies.

There is no version pin for `zc.buildout` as it has to be installed in the virtual environment but `checkversions` also prints its version number.

There is no version pin for `zc.recipe.egg` in `versions-prod.cfg` as it is only needed for buildout install and not for pip, so we do not want to have it in `requirements.txt`.

The script is called two times so the rendered version updates can be easily assigned to the correct file.

• Announce the release to the world via `zope-announce@zope.org` and https://community.plone.org/c/announcements.

### 5.2 Maintaining the Zope documentation

#### 5.2.1 Contributing to the documentation

Any signed Zope contributor may contribute to the Sphinx-based documentation in the `docs` subfolder, including *The Zope Book* and the *Zope Developer’s guide*.

Just like with code contributions, please follow best practice. Test your changes locally before creating a pull request or pushing to the repository. Use a reasonable line length (<80).

#### 5.2.2 Building the documentation

After you have bootstrapped and run the buildout, you can build the documentation using the script `bin/make-docs` to create the documentation HTML output. The script will tell you where it saves the output.

#### 5.2.3 The official documentation site on *Read the Docs*

Pushes to the Zope repository on GitHub will automatically trigger an automatic documentation refresh on the official documentation site at https://zope.readthedocs.io. This is true for the `master` branch, but also for versions 2.12 and 2.13. The trigger is implemented as a GitHub Webhook, see *Settings | Webhooks* in the GitHub repository.

The RTD configuration at https://readthedocs.org/projects/zope/ is currently maintained by the following people:

- Hanno Schlichting
- Michael Howitz
- Tres Seaver
- Jens Vagelpohl
These are all the changes for Zope 5, starting with the alpha releases.
The change log for the previous version, Zope 4, is at https://github.com/zopefoundation/Zope/blob/4.x/CHANGES.rst

6.1 5.0a1 (unreleased)

• Remove deprecated `postProcessInputs` request method (#782)
• Clean up and sanitize permissions used for WebDAV-related methods
  • Fix sort link URLs on `manage_main` (#748)
  • More tests to make sure all `__str__` implementations return native strings (#692)
• Restore WebDAV support in Zope (#744)
• Fix longstanding test bug by forcing the page template engine. Many tests in `Products.PageTemplates` used the old Zope page template engine because the correct one was not registered during setup.
• Remove deprecated module `ZPublisher.maybe_lock` (#758)
• Remove Help System methods from the product context (#756)
• Remove more deprecated code (#757)
• Close opened db during shutdown (as ZServer is already doing). (#740)
• Updated Zope documentation sources for Zope 5 (#659)
• The method `unrestrictedTraverse` raises an error when the argument `path` is not something it can work with (#674)
• Improve ZMI Security Tab usability for high numbers of roles (#730)
• Some small ZMI rendering fixes (#729)
• Fix error when using database minimize in the ZMI (#726)
• Fix \_\_getattr\_\_ signature in UnauthorizedBinding (#703)
• Remove more Python 2 support code (#692)
• Move retried request delay handling out of supports\_retry (#474)
• Remove all backwards-compatibility code marked to go away in Zope 5 (#478)
• Fix VirtualHostMonster not being able to set mappings under Python 3. (#708)
• Reduce the danger of acquiring built-in names on the ZMI Find tab (#712)
• Restore the mistakenly removed Properties ZMI tab on Image objects (#706)
• Fix OFS.Image.File.\_\_str\_\_ for Pdata contents (#711)
• Add wsgi.file_wrapper implementation https://www.python.org/dev/peps/pep-0333/#optional-platform-specific-file-handling (#719)
• Set REMOTE\_USER in wsgi environ using Zope user authentication (#713)
• Improve documentation for Zope’s error logging services.
• Add Paste as extras_require dependency to pull in Paste when installing with pip and constraints.txt to prevent startup errors. This requires adding the [wsgi] extra in the egg specification. (#734)

6.1.1 Backwards incompatible changes

• Drop support for Python 2.7 aka Zope 5 cannot be run on Python 2 any more. If you are still running on Python 2.7 upgrade to the latest Zope 4 version first, migrate to Python 3 and than switch to Zope 5. (#692)
• Remove Python 2 support code and dependency on six. (#692)
• Drop support for running Zope with ZServer as it is Python 2 only. (#592)
Welcome to *The Zope Book*. This book is designed to introduce you to Zope, an open-source web application server.

### 7.1 Preface

Welcome to *The Zope Book*. This book is designed to introduce you to Zope2, an open-source web application server. To make effective use of the book, you should know how to use a web browser and have a basic understanding of the Hyper Text Markup Language (HTML) and Uniform Resource Locators (URLs). You don’t need to be a highly-skilled programmer in order to use Zope2, but you may find the understanding of some programming concepts (particularly in object-oriented programming) to be extremely helpful.

#### 7.1.1 Preface to the 2.12 edition

This book has originally been written for Zope 2.6 back in 2002. It has been available in an almost unmodified form for the last seven years. During those many years quite a bit has happened in Zope itself and the general web market. The 2.12 edition of this book does not try to write a new book on how-to do Zope development today. Instead it tries to update the original books content to be true and helpful again. Many of the underlying principles of Zope2 have not changed in the last years. The ZMI, security machinery, page templates and how-to use the ZCatalog are still there in an almost unmodified fashion. The general ideas behind object orientation, being Python based and the general architecture are still the same.
If you want to understand Zope2 you still need to understand how Acquisition works, even though it has been discour-aged as a way to design your application logic.

One of the most notable differences between the original Zope2 approach and today’s best-practices is in the way you develop applications with Zope2. The original Zope2 approach has focussed on a Through-The-Web (TTW) development model. You would create your entire application and manage your data through the same browser interface and store everything transparently in the same database. This model has worked very well in the beginning of “the web” as many dynamic websites have been rather simple and specialized projects.

Over the years websites have grown their requirements and often turned into development projects of a considerable size. Today websites are understood as applications in themselves and need an approach which is no longer compatible with the TTW approach of the early Zope2.

In this book you will still read about using the TTW approach for many of the examples. Please understand this as a way to quickly and easily learn about the underlying technologies. If you want to built an application based on top of Zope2, you are almost always better of approaching the project from the so called “file-system based approach” or using Python packages to extend Zope in a predictable way.

### 7.1.2 How the Book Is Organized

This book is laid out in the following chapters:

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

- **Zope Concepts and Architecture**
  
  This chapter explains fundamental Zope concepts and describes the basics about Zope’s architecture.

- **Installing and Starting Zope**
  
  This chapter explains how to install and start Zope for the first time. By the end of this chapter, you will have Zope installed and working.

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

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

- **Using Basic Zope Objects**
  
  This chapter introduces *objects*, which are the most important elements of Zope. You’ll learn the basic Zope objects: content objects, presentation objects, and logic objects, and you’ll build a simple application using these objects.

- **Acquisition**
  
  This chapter introduces *Acquisition*, which is Zope’s mechanism for sharing site behavior and content.

- **Basic Zope Scripting**
  
  This chapter will introduce you to the basics of scripting.

- **Using Zope Page Templates**
This chapter introduces Zope Page Templates, another Zope tool used to create dynamic web pages. You will learn about basic template statements that let you insert dynamic content, and how to create and edit page templates.

• Creating Basic Zope Applications

This chapter presents several real-world examples of building a Zope application. You’ll learn how to use basic Zope objects and how they can work together to form basic applications.

• Users and Security

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

• Advanced Page Templates

This chapter goes into more depth with Zope Page Templates. You will learn all about template statements, expression types, and macros, which let you reuse presentation elements.

• Advanced Zope Scripting

This chapter covers scripting Zope with Python. You will learn how to write business logic in Zope using tools more powerful than TAL, about the idea of scripts in Zope, and about Scripts (Python).

• Zope Services

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

• Basic DTML

This chapter introduces DTML, the second tag-based scripting language. You’ll learn DTML syntax, its basic tags, and how to use DTML templates and scripting facilities. After reading this chapter, you’ll be able to create dynamic web pages with DTML.

• Advanced DTML

This chapter takes a closer look at DTML. You’ll learn about DTML security, the tricky issue of how variables are looked up in DTML, advanced use of basic tags, and the myriad of special purpose tags.

• Searching and Categorizing Content

This chapter shows you how to index and search objects with Zope’s built-in search engine: the Catalog. You’ll learn about indexing concepts, different patterns for indexing and searching, metadata, and search results.

• Relational Database Connectivity

This chapter describes how Zope connects to external relational databases. You’ll learn about features that allow you to treat relational data as though it were Zope objects, and security and performance considerations.

• Virtual Hosting Services

This chapter explains how to set up Zope in a “virtual hosting” environment, in which Zope subfolders can be served as “top-level” host names. It includes examples that allow virtual hosting to be performed either “natively” or using Apache’s ‘mod_rewrite’ facility.

• Sessions

This chapter describes Zope’s “sessioning” services, which allow Zope developers to “keep state” between HTTP requests.

• 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. You’ll learn about the tools and techniques needed to turn a small site into a large-scale site, servicing many simultaneous visitors.

• Managing Zope Objects Using External Tools
  This chapter explains how to use tools outside of your web browser to manipulate Zope objects.

• Maintaining Zope
  This chapter covers Zope maintenance and administration tasks, such as database “packing” and package installation.

• Appendix A: DTML Reference
  Reference of DTML syntax and commands.

• Appendix B: API Reference
  Reference of Zope object APIs.

• Appendix C: Page Template Reference
  Reference of Zope Page Template syntax and commands.

• Appendix D: Zope Resources
  Reference of “resources” which can be used to further enhance your Zope learning experience.

• Appendix E:
  DTML Name Lookup Rules Describes DTML’s name lookup rules.

7.2 Introducing Zope

Attention: This document was written for Zope 2.

Zope is a family of related Python packages focussed on web technologies. The first version of Zope has originated from a company called Zope Corporation.

Today the Zope Foundation holds the copyright of the Zope source code and supervises a diverse community of open-source contributors working on a variety of related projects.

This book is about the original Zope project, today known as Zope2. When we refer to Zope in this book without a narrower specification we speak of Zope2.

Other projects include the Zope3 web application framework, many individual packages located in the Zope Subversion Repository and projects being based or related to these packages like Grok and Repoze. One of the more widely known applications based on top of Zope2 is a content management system called Plone.

Zope2 itself is a web framework that allows developers of varying skill levels to build web applications. This chapter explains Zope’s purpose, what problems it solves and what audience it targets in greater detail. It also describes what makes Zope different and more powerful than similar applications.

NOTE: The moniker “Zope” stands for the Z Object Publishing Environment (the “Z” doesn’t really mean anything in particular).
7.2.1 The Static Web Site Dilemma

When a company or organization goes through the process of developing and eventually deploying a website, one of its most important goals is to present timely and up-to-date information to its website visitors.

Let us consider two examples of such time-dependent sites:

- a stock market information site that needs to be updated with new information continually, maybe as often as every five or 10 minutes. It will also present information tailored to each visitor’s preferred settings (portfolios, stocks to follow, etc.)

- a commercial website that helps its visitors sell and buy used automobiles. It is usually required that such a site run advertisements only for cars that have not yet been sold. It is also important that new ads be posted immediately after they’ve been placed by a seller.

These two examples describe two very different sites that nevertheless have one basic requirement in common: automated and periodic updates of the information presented. If this single requirement is not met, these sites will likely be unsuccessful.

So, how does Zope work to fulfill such a requirement? To understand this, we need to consider how websites are perceived by their visitors and the basic ways in which websites can be constructed.

In general, many website visitors think about navigation in terms of moving “from page-to-page” within a website. When they click a hyperlink, their browser transports them to a new page. When they hit their browser’s back button, they are returned to the last page they visited, and so on.

Some websites are static. A static website stores its information in files on a web server. Each file then represents a complete page on the website. This may seem like a simple and efficient way of creating a website; however, updating the information within those pages becomes a problem when the site consists of more than a few pages, and the pages, or parts of the pages, need to be updated frequently.

The layout of text and images that are displayed in a user’s web browser when the user visits a website are commonly composed in 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 formatting in HTML is transferred between the website and the user’s browser. The browser interprets the chunk of text and displays text and images to the user. The chunk of text which is transferred is typically referred to as a page.

To achieve this, the static website requires a person with a privileged level of access (sometimes termed the webmaster) to manually create and update the site’s content.

Typically, this is done by editing a set of text-based files on the web server (the machine that runs the website), where each file represents a single page. In some cases, a site-wide change to the “look-and-feel” of a static website requires that the webmaster visit and update each and every file that comprises the website.

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 that have expired and create new pages for ads that have been recently sold. He then needs to make sure that no hyperlinks on other pages point to any of these deleted pages.

Obviously, this quickly becomes a lot of work. With any more than a few pages to update each day, this type of repetitive work can become pretty dull. In addition, being a human being, the webmaster may also make mistakes, such as forgetting to update or remove critical pages. While updating a static website with only 10 to 20 pages might be dull, it’s perfectly manageable. However, websites can typically grow to encompass thousands of files, making the process of “timely updates” a non-trivial (and sometimes impossible) task.

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 web server computer could automatically do a lot of the work I now do manually.” And he would be right.
At this point, the webmaster is ready to be introduced to web applications. It is in this area where Zope’s strength and power becomes clear.

### 7.2.2 What Is A Web Application?

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 a web server that fronts a statically-built website and one that fronts a web application. But unlike a static website, a web application creates its “pages” *dynamically*, or on-the-fly, upon request. A website that is dynamically-constructed uses a computer program to provide its content. These kinds of dynamic applications can be written in any number of computer languages.

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 Gmail; or view the latest news ala CNN.com.

In a dynamically-constructed website, the webmaster is not required to visit the site “page-by-page” in order to update its content or style. Instead, he is able to instruct the web server to *generate the site’s HTML pages dynamically*, where each page is made up of different bits of content. While each bit of content is unique, each can nevertheless appear in several pages if so instructed by the web server. In this way, the webmaster is able to create a common “look and feel” for the set of pages that make up his site. The software on the web server that generates these pages is the web application.

If our auto-classifieds webmaster chose to construct a web application to maintain his classifieds system, he could maintain a list of “current” ads separate from the HTML pages, 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.

A framework that 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 WebSphere, JBoss, and (to some extent) SAP NetWeaver.

Zope is a web application server, which is not a web application in itself; rather it is framework that allows people to construct web applications. Sometimes this framework is called an **application server**.

Using some common computer programming language, an application server typically allows a developer to create a web application, but it also provides services beyond the basic capabilities of the programming language used. Examples of such services are web page template creation facilities, a common security model, data persistence, sessions, and other features that people find useful when constructing a typical web application.

### 7.2.3 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 with an application server framework (as opposed to a “raw” computer language, such as Java, Perl, Python, or C), you are able to utilize the services of the framework that have already been written and proven to work, and you avoid the need to write the functionality yourself “from scratch” in a “raw” language.

Many application servers allow you to perform some of the following 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 and typically come with facilities for personalization, database integration, content indexing, and searching.

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 *content management system* allows non-technical 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. You must 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 must be able to accept requests from other computer programs. For example, if you're building a news site, you may wish to share your news stories with another site; 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 offer methods for 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 sites.

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

### 7.2.4 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 by a number of Zope features:

- Zope is free of cost and 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 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 packages 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.

- Applications created in Zope can scale almost linearly using Zope’s built-in “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 provides a granular and extensible security framework. You can easily integrate Zope with diverse authentication and authorization systems, such as LDAP, Kerberos, and RADIUS, simultaneously and using pre-built modules. Many other application servers lack support for important authentication and authorization systems.
Zope runs on most popular microcomputer operating system platforms: Linux, Windows, Solaris, FreeBSD, NetBSD, OpenBSD, and Mac OS X. 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 that 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.

### 7.2.5 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.
- **Integrators** integrate the software written by component developers and native application server services, building an application in the process.
- **Web 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**, **integrators**, and **web designers**. These three groups can collaborate to produce an application using Zope’s native services and third-party Zope Plugins. They typically produce applications useful to **content managers** and **consumers** under the guide of the **information architect**. **Administrators** 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 blog, a content management system, or a “e-shop-in-a-box” application.

However, freely available Plugins built on top of Zope offer these kinds of services. At the time of this writing, the Python Package Index lists roughly 400 Plugins that you can browse and even reuse in your own applications. These include Plugins for blogging, content management, internationalization, and e-commerce.

Zope is not a visual design tool. Tools like Macromedia Dreamweaver and Adobe GoLive allow designers to create “look and feel”. You may use these tools to successfully manage Zope-based web sites, but Zope itself does not replace them. You can edit content “through the web” using Zope, but it does not try to replace the features offered by these kind of tools.

### 7.2.6 Introduction to Zope Maintenance and The Zope Community

A community of developers is responsible for maintaining and extending the Zope application server. Many community members are professional consultants, developers, and webmasters 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.

The Zope Foundation controls the distribution of the defacto, “canonical”, official Zope version, and permits its developers, as well as other selected developers, to modify the distribution’s source code.
The Zope community gets together occasionally at conferences, but it commonly discusses all things 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 its revenue 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.

7.2.7 Zope’s Terms of Use and License

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 Foundation.

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, when you redistribute a GPL-licensed application, you must distribute it under the terms of the GPL, including licensing any modifications or extensions you make under the GPL. You must also provide the full source code, including source for your modifications.

However, this is not required for ZPL-licensed applications. You may modify and redistribute 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.

7.2.8 Zope History

In 1996, Jim Fulton (the current CTO of Zope Corporation, the orginators 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 developed a commercial application server based on their three open-source components. This product was called Principia. In November of 1998, investor Hadar Pedhazur convinced Digital Creations to open source Principia. These packages have evolved into what today are the core components of Zope.

Most of Zope is written in the Python scripting language, with performance-critical pieces written in C.

7.3 Zope Concepts and Architecture

**Attention:** This document was written for Zope 2.
7.3.1 Fundamental Zope Concepts

The Zope framework has several fundamental underlying concepts, each of which should be understood in order 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 one to focus instead 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, and to write web application logic in the Python language. Zope also comes with one solution that allow you to “template” text, XML, and HTML: Zope Page Templates (ZPT).

Object Orientation

Unlike common, file-based web template 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 Python. The concept of object orientation may take a little “getting-used-to” if you’re an old hand at procedural languages used for web scripting, such as Perl or PHP. However, you will easily grasp its main concepts by reading the Object Orientation chapter, and by trying the hands-on examples in this 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 to request a response.

Zope’s object structure is hierarchical, which means that a typical Zope site is composed of objects that 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:

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 into files and directories on a file system. Zope similarly maps URLs to objects in its object database.

A Zope object’s URL is based on its path, which 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 would 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’. 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 to be 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 can use your Web browser to access the Zope management interface (ZMI). Basic management and application configuration tasks can be done completely through the Web using only a browser. The ZMI provides a familiar, Windows Explorer-like view of the Zope object system.

Any object in the object hierarchy can be configured. 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. For example a “Database Connection” Zope object provides views that let you modify its connection string or caching parameters. All objects also have a “Security” view that allows you to manage their individual access control settings.

Zope had a much larger focus on Through-The-Web activities in its beginning. In recent years the Through-The-Web model has been discouraged for any kind of development and reduced to configuration tasks.

### 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 with not only the Web object model, but also the Web development model. Today’s successful web applications require the participation of many people across an organization with 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 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 user account 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 robust set of possible permissions, richer than that of a conventional file-based system. Permissions vary by object type, based on the capabilities of that object, which enables the implementation of fine-grained access control. For example, you can set access control so that content managers can use “SQL Method” objects without being able to 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 “Page Templates,” 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 sub-folder, essentially allowing you to delegate the creation and user management of subsections of your website to semi-trusted users, without having to worry about those users changing the objects “above” their own folder.
Native Object Persistence and Transactions

By default, 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 ZODB. If an error occurs in your application during a request, any changes made during the request will be automatically rolled back. The ZODB 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, when used with Zope, can also play in Zope’s transactional framework.

Acquisition

One more prominent aspect of Zope is acquisition, whose core concepts are 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 sub-folders can use it through acquisition. If the query needs to be changed, you can change it in one place without worrying about all of the sub-objects that use the same query.

If you are familiar with Cascading Style Sheets (CSS), you already know how an element in an HTML document can inherit cascading properties from its parent or ancestor elements. Containment acquisition works in the same fashion: if a document X is contained in folder Y, document X can access the attributes of folder Y through acquisition. Note that some advanced aspects of acquisition may break this analogy; these are discussed in the Advanced Zope Scripting chapter.

Acquisition is explained in further detail in the chapter on Acquisition.

Zope Is Extensible

Zope is highly extensible, and component developers can create new types of Zope objects by writing new Zope add-on in Python. The Zope software provides a number of useful, built-in components to aid extension authors in development, 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-ons are available that provide features like drop-in web discussion topics, desktop data publishing, XML tools, and e-commerce integration. Many of these add-ons have been written by highly active members of the Zope community, and most are also open source.

7.3.2 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 following figure and explained below:
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).

**ZServer**

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 web server. Zope works with these servers also, and any other web server that supports the Common Gateway Interface (CGI). In production environments, it can be advantageous to run a server like Apache or Squid “in front of” Zope in order to help sanitize incoming requests, augment its capabilities (e.g., terminate HTTPS connections), and cache Zope-provided content.

**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 web server. Zope works with these servers also, and any other web server that supports the Common Gateway Interface (CGI). In production environments, it can be advantageous to run a server like Apache or Squid “in front of” Zope in order to help sanitize incoming requests, augment its capabilities (e.g., terminate HTTPS connections), and cache Zope-provided content.
Zope Core

This is the engine that coordinates Zope activity, driving its management interface and object database.

Object Database

When you work with Zope, you are usually working with objects that are stored in the ZODB.

Relational database

You don’t have to store your information in Zope’s object database if you don’t want to. Zope also works with other relational databases, including Oracle, PostgreSQL, Sybase, and MySQL.

File System

Zope can, of course, work with documents and other files stored on your server’s file system.

Products

Zope also allows site managers to add new, pre-built object types to Zope by installing add-ons on the Zope server file system. These are referred to as Products or Add-ons. Technically they are normal Python packages.

7.4 Installing and Starting Zope

Attention: This document was written for Zope 2.

By the end of this chapter, you should be able to install and start Zope. It is fairly easy to install Zope on most platforms, and it typically takes no longer than ten minutes to complete an installation.

7.4.1 Downloading Zope

There are typically two types of Zope releases: a “stable” release and a “development” release. 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, from which the most recent stable and development versions are always available in the Download area.

Zope comes as a “binary” release for the Windows platform, and in source format for UNIX-like operating systems. 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, HPUX, IRIX, DEC OFS/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 if you have a C compiler and associated development utilities, then it is highly likely that you will be able to compile Zope. A notable exception is Mac OS between versions 7 through 9, as Zope does not run at all on these platforms.
7.4.2 Installing Zope

Zope’s installation steps vary somewhat, depending on your operating system platform. The sections below detail installing the binary version of Zope on Windows on Intel platforms, and a source installation on Linux.

Installing Zope for Windows With Binaries from Zope.org


![Fig. 1: Current stable Zope release for Windows](image)

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 into which you downloaded the file, and double-click on the file’s icon. The installer then begins to walk you through the installation process.

![Fig. 2: Zope installer](image)

Click Next. The installer asks for an installation path. The default is usually acceptable, though you are, of course, free to choose another path. Then click Next. You then can choose which components to install.

You should select “Full installation” unless you have previously installed Zope and know what you are doing. On the next screen, you may customize the entry placed in your Start Menu folder. Click Next again. The installer now asks you whether you would like to run Zope as a service, unless you are running Windows 98 or ME, on which such services are not available. If you are only running Zope for personal use, there is no need to run it as a service.

Upon clicking Next, the installer takes you to the “Instance Setup” Screen.
Fig. 3: Beginning the installer
Select Components

Which components should be installed?

Select the components you want to install; clear the components you do not want to install. Click Next when you are ready to continue.

- [ ] Main Zope files
- [x] Zope 'instance home' files (required to run Zope immediately) 1.4 MB

Current selection requires at least 30.1 MB of disk space.

Fig. 4: Select components
Select Additional Tasks
Which additional tasks should be performed?

Select the additional tasks you would like Setup to perform while installing Zope 2.7.0, then click Next.

- Run your Zope instance as a Windows service [start Zope automatically at system startup]

Fig. 5: Server options
Fig. 6: Instance setup
You can have more than one Zope running on your PC, but each has to have its own *Instance Home*, which is the path to specify here. This path is where Zope will later place its database files. Make sure that you have enough disk space left on the specified drive and that you can make backups easily.

The *Next* screen asks you for a password for an initial administrative account. You use this account to log in for the first time and create more users. Note that the installer does not ask you to verify your password, so be careful not to mis-type it.

Click *Next* after entering a password. The installer presents an overview, form which you can commence installation by clicking *Install*. After a few moments, the Zope installer will present you with a “Completion” screen.

Let the installer start Zope for you, or start Zope manually by navigating to the Zope folder in the Start Menu and selecting “Run Zope in Console”. See the section below entitled *Starting Zope*.

### 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 that you have a “C” compiler on your system (*GNU gcc* is preferred);

- ensure that you have a recent “make” on your system (*GNU make* is preferred);
Completing the Zope 2.7.0 Setup Wizard

Setup has finished installing Zope 2.7.0 on your computer. The application may be launched by selecting the installed icons.

Click Finish to exit Setup.

- View ‘QuickStart’ document in a web browser
- Launch Zope

Fig. 8: Installation completion
• install the Python language on your system from source, or install a binary Python package, including development headers.

Zope is written primarily in the Python language, and Zope requires Python in order to be able to run at all. While binary versions of Zope ship with a recent Python version, the source Zope distribution does not. Zope developers try to use the most recent Python for Zope, but often the latest Python version is more recent than the officially-supported Zope version. Zope 2.12 requires Python 2.5.4 or later, and Zope versions 2.11 and 2.10 require a Python 2.4.*x* version equal to or greater than 2.4.3. For the most recent information on which version of Python is required for compiling Zope, see the release notes on the release Web page.

You can obtain detailed instructions for downloading, compiling, and installing Python from source at the Python.org website. Most Linux distributions ship with a pre-installed Python 2.5, but care is required 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 may be installed separately, but sometimes they are not. The binary packages that ship with Debian have been used with some level of success, but it is generally advisable to compile and install Python from source if you wish to also 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.

Download the source to your home, or some other directory, `cd` to that directory, and unpack it with something similar to:

```
$ mkdir ~/myzope
$ cd ~/myzope
$ gunzip -c /tmp/Zope-*.tgz | tar xvf -
```

where * represents the Zope release version of the source tarball.

Zope now uses the conventional UNIX build sequence: configure, make, make install.

To configure Zope, `cd` to the Zope directory and issue the configure command:

```
$ cd Zope-*
$ ./configure --prefix=/where/to/install/zope
```

Replace `/where/to/install/zope` above with an appropriate path, such as `~/myzope/zope2`. This path is referred to as the ZOPE_HOME. If you want to install Zope in a system directory instead of your user home, replace `~/myzope/zope2` with an appropriate path, e.g., `/usr/local/zope2`, and make sure that you have suitable privileges for installing and starting Zope (‘sudo’ or ‘root’).

If the configure script is unable to find your Python installation, it will report an error not unlike this one:

```
$ ./configure --prefix=~myzope/zope2

Configuring Zope installation
Testing for an acceptable Python interpreter...

No suitable Python version found. You should install Python version 2.5.4 before continuing. Versions 2.6.1 2.6.0 also work, but not as optimally.
```

In this case, you must point the installer to your Python interpreter, which you should have installed previously, either from a binary package or compiled from source.

Use the `--with-python` option to the configure script, e.g., for a python living under `/usr/local`:

```
$ ./configure --prefix=~myzope/zope2 \
--with-python=/usr/local/bin/python
```
Replace `/usr/local/bin/python` with the path to your Python executable.

Zope is now ready to be built. From within the source directory, issue:

```bash
$ make
[ lots of output snipped ]
Zope built. Next, do 'make install' (or 'make instance'
to run a Zope instance directly from the build directory).
```

You are now ready to install Zope. To do this, you will have to execute ‘make install’

```bash
$ make install
[ lots of output snipped ]
Zope binaries installed successfully.
Now run '~/myzope/zope2/bin/mkzopeinstance.py'
```

With the Zope binaries installed, you are now ready to create a **Zope instance**, which holds configuration and runtime data for a single Zope server process. This helps keep your own or third-party software separate from the main Zope source.

Assuming that you want to install a Zope instance in the directory `~/myzope/instance`, in order to create a Zope instance, you would run the following command:

```bash
$ ~/myzope/zope2/bin/mkzopeinstance.py
```

You will need to provide the following values:

- The directory where your instance should be located, or the `INSTANCE_HOME`. The instance home will hold your database files, log files, configuration files, and scripts to start and stop the instance. For our example, we assume the instance home to be located at `~/myzope/instance`.

- Username and Password for an initial Zope user. You will log in with this username and password to create your own Zope users. To change the username or password for your initial Zope user, run:

```bash
$ cd ~/myzope/instance
$ ~/myzope/zope2/bin/zpasswd.py inituser
```

You will have to provide the username and password you wish to set; optionally, you can specify the hashing method and an additional domain restriction.

Zope installation is now complete. Read on to see how to start your brand-new Zope.

### 7.4.3 Starting Zope

Zope is managed via a web browser, and Zope contains its own web server (called **ZServer**). A successful Zope startup implies that Zope’s web server starts, which allows you to access the Zope management interface (ZMI) via your web browser. You can access the ZMI 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 requests on TCP port 8080. If your Zope instance fails to start, make sure that another application isn’t already running on the same 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 that indicate that Zope is listening on ports other than the default 8080 HTTP, don’t panic: it’s likely just one of these additional ports.
7.4.4 Using Zope With a Web Server

Zope works with **NGINX**, **Apache** and other popular web servers.

The Virtual Hosting Services chapter of this book provides rudimentary setup information for configuring Zope behind Apache.

If you are just getting started with Zope, note that it is not necessary to configure a web server to serve your Zope pages, as the standard Zope setup comes with the WSGI server **Waitress**.

You typically only need to configure a web server if you want to use it to serve Zope pages in a production environment.

7.4.5 Starting Zope on Windows

If you’ve installed Zope to “run manually” (as opposed to installing Zope as a “service”), navigate to the Zope folder in your Start Menu and click on **Run Zope in Console**. A console window with process startup information will be displayed.

If you 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 instance started as a service writes events to the standard Windows Event Log; you can keep track of the Zope service’s start and stop events by reviewing the Event Log. A Zope instance which has been installed as a “service” can also be run manually by invoking the **Run Zope in Console** menu entry as described earlier. Take care not to run Zope manually and as a service at one time: make sure to stop the Zope service first before starting it manually.

7.4.6 Starting Zope on UNIX

To start your Zope instance (which we assume lives in `~/myzope/instance`), issue the command:

```
$ ~/myzope/instance/bin/zopectl start
```

This will start the instance in the background. Alternatively, you can start it in the foreground and watch its progress by issuing the command:

```
$ ~/myzope/instance/bin/zopectl fg
```

Run the **zopectl** script with a parameter of `help` to get a list of additional commands:

```
$ ~/myzope/instance/bin/zopectl help
```

Starting Zope as the Root User

**ZServer** (Zope’s server) supports **setuid()** on POSIX systems in order to be able to listen on low-numbered 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.

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, because if any other daemon on a system that ran as nobody were to be compromised, this would open up your Zope object data to vulnerability.

If you do need to have ZServer listening on low ports, you will need to start **zopectl** as the root user, and to specify what user ZServer should **setuid()** to. This can be done by setting the **effective-user** parameter in your
Zope instances configuration file, residing in $INSTANCE_HOME/etc/zope.conf, and by making sure that the log and database files are writeable by this user.

### 7.4.7 Your Zope Installation

To use and manage Zope, you will need a web browser. Start a web browser on the same machine on which you installed Zope, and browse to the URL http://localhost:8080/.

If your Zope instance has been properly installed, and you’re visiting the correct URL, you will be presented with the Zope “QuickStart” screen.

![Zope Quick Start](image)

Fig. 9: Zope QuickStart

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

### 7.4.8 Logging In

For some of the tasks you want to do with Zope, you need to use its management interface: the ZMI. To log into the ZMI, 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 via this URL will result in an authentication dialog, into which you can enter the “initial” username and password you chose when you installed Zope. You will then be presented with the ZMI.

If you do not see an authentication dialog and the ZMI, refer to the Troubleshooting and Caveats section of this chapter.
7.4.9 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.

The Control Panel displays information about your Zope, such as the Zope version you are running, the Python version that Zope is using, the system platform, the INSTANCE_HOME, the CLIENT_HOME, Zope’s process id, the network services that have been started, how long Zope has been running for, and other installation specifics. 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 Restart button in the Control Panel. Clicking Restart will cause Zope to shut down and then immediately start back up again. It may take Zope a few seconds to come back up and start handling requests. You don’t need to shut your web browser down and restart it to resume using Zope after pressing Restart, as the page refreshes automatically; just wait for the Control Panel display to reappear.

To shut Zope down from the ZMI, click Shutdown. 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 as needed.

7.4.10 Controlling the Zope Process from the Command Line

- If you started Zope in the foreground, press “Ctrl+C” in the terminal window from which you started Zope.
- If you started Zope in the background, use the zopectl script:

  $ ~/myzope/instance/bin/zopectl stop
7.4.11 Customizing your Zope instance

Zope’s configuration is done via the file ‘$INSTANCE_HOME/etc/zope.conf’. This contains numerous configuration directives for customization.

The zope.conf file features extensive inline documentation, which we will not reproduce here. Instead, we will give an overview and some additional hints for the most-widely used directives:

**Server stanzas and port-base**

The port-base directive, together with stanzas for the individual servers, determine the ports on which specific servers listen for incoming Zope requests. The stanzas are formed with XML-like constructs:

```xml
<http-server>
  # valid keys are "address" and "force-connection-close"
  address 8080
</http-server>
<ftp-server>
  ...
</ftp-server>
<webdav-source-server>
  ...
</webdav-source-server>
```

- On Unix use the “kill” command against the process id in the “var/Z2.pid” file inside the Zope instance directory:

```bash
$ kill `cat var/Z2.pid`
```
The `address` directive determines the port on which the respective server listens. The HTTP Server in this example listens on port 8080.

The `port-base` directive comes in handy if you want to run several Zope instances on one machine. `port-base` specifies an offset to the port on which all servers listen. Let us assume that our HTTP Server’s ‘address’ directive is set to 8080, as in our example above, and ‘port-base’ is specified as 1000. The port on which the HTTP server will listen, will be the `address` value of 8080, plus the `port-base` offset value of 1000, or 9080. Assuming the FTP server’s `address` directive is set to 8021, the FTP Server will then listen on port 9021, and so on.

**The `debug-mode` directive**

This directive is a switch, specified as either `on` or `off`. When set to `on` (the default), Zope runs in debug mode, which causes Zope to reload file system-based templates, and several other settings suitable for development, in real time. In a production environment, to reduce unnecessary overhead, you should ensure that this directive is set to `off` unless you are actively troubleshooting a problem.

**Switch the User the Zope process runs as: `effective-user`**

This directive causes Zope to `setuid(2)` to the specified user when run as root on a UNIX system. This method boosts system security, as a compromised Zope instance would not enable a compromised user to damage easily an entire system. One motivation for running Zope as root in the first place is to be able to bind to privileged ports, or ports with values below 1024.

**Logging**

Three log facilities are provided:

- **Access logging** logs individual HTTP Requests in a common format, by default to the file `log/Z2.log` in your instance home.
- **Event logging** logs Zope events, such as start and stop information and debugging messages.
- **Trace logging** logs detailed Zope debugging information.

Each log message has an associated severity level, ranging from `CRITICAL`, `ERROR`, `WARN`, and `INFO`, to `DEBUG` and `ALL`. You can specify a filter for log messages with the `level` directive inside a logger stanza. Set the level to `ALL` to get all log messages, or to `ERROR` or `CRITICAL` to see only the most serious messages.

Although the default is to write the messages to a log file, you can instead arrange for log messages to be mailed to you, or to go to `syslog(3)` (on UNIX) or the event log (on MS Windows).

For further documentation, see the inline comments in `zope.conf`.

### 7.4.12 Troubleshooting and Caveats

#### Browser cannot connect to port 8080

If your browser fails to connect with anything on TCP port 8080, your Zope instance may be running on a non-standard TCP port (for example, some versions of Debian Linux ship with Zope’s default TCP port as 9673). To find out exactly which URL to use, look at the logging information Zope prints as it starts up when started in the foreground, i.e., when started with `./runzope` or `./zopectl fg`. For example:
The first log entry indicates that Zope’s web server is listening on port 9673 on host arod. This means that the management URL is http://arod:9673/manage.

As mentioned previously, Zope only prints to the console when started in the foreground, with ./runzope or runzope.bat. This logging information can be found in the log/event.log file in your INSTANCE_HOME directory.

**Forgot administrative password**

If you forget or lose your initial Zope user name and password, shut Zope down, 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.

**When All Else Fails**

If there’s a problem with your installation that you just cannot solve, do not despair. You have many places to turn for help, including the Zope mailing lists 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 mailing list 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 and able to help you solve the problem.

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

### 7.5 Object Orientation

**Attention:** This document was written for Zope 2.

To make the best use of Zope, you will need a grasp on the concept of *object orientation*, which is a software development pattern used in many programming languages (C++, Java, Python and others) and computer systems that 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.
7.5.1 Objects

In Zope, as in other object-oriented systems, your application is designed around objects, or self-contained “bundles” of data and logic. It is easiest to describe these bundles by comparing them to other programming concepts.

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

- Code. For example, a typical CGI-based web application may have a bit of logic in the form of a PHP script, which retrieves employee data from a database and displays tabular data to a user.
- Data. For example, you may have employee data stored in a database, such as MySQL or Oracle, on which some code performs read or change operations. This data exists almost solely for the purpose of the code that operates upon it; without this code, the data holds little to no value.

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

- Objects. Simply stated, these 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. However, the object will also contain “logic,” or code, that can manipulate and display its data.

In a non-object-oriented application, your data is kept separate from your code. But in an object-oriented application, both your data and your code are stored in one or more objects, each of which represents a particular “thing”. These objects can represent just about anything. In Zope, the Control_Panel is an object, Folders that you create are objects, and 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 add-ons define their own types of objects, which are then entered in to the Zope “add list” so that you can create objects based on them. An add-on author might define a “Form” object or a “Weblog” object. Basically, anything that can be defined using a noun can be modeled as a Zope object.

As a programming methodology, object orientation allows software developers to design and create programs in terms of “real-world” things, such as 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 that 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 the real-world objects, like Employees and Forms, that we can understand more readily and quickly.

The concept of abstraction also encourages programmers to break up a larger problem by addressing the problem as smaller, more independent “sub-problems,” which allows developers to define and address solutions in much smaller, more feasible terms. When you design an application in terms of objects, they become the pieces that eventually define the solution to all the “sub-problems” of a particular “big” problem.

7.5.2 Attributes

An object’s data is defined by its attributes, or pieces of data that describe aspects of the object. For example, an attribute of an Employee object might be called “phone_number,” which might contain a series of characters that represent the employee’s phone number. Other attributes of an Employee object might be “first_name,” “last_name”, and “job_title,” all of which give additional, detailed information about each Employee.

It may help to think of the set of attributes belonging to an object as a sort of “mini-database” that contains information representing the “real-world thing” that the object is attempting to describe. The complete collection of attributes assigned to an object defines that object’s state. When one or more of an object’s attributes are modified, the object is said to have changed its state.
7.5.3 Methods

The set of actions that an object may perform is defined by its methods. Methods are code definitions attached to an object that perform actions based on the object’s attributes. 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 a value of “Vice President” or “Janitor,” depending on which Employee object is being queried.

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, or attached to, an object: instead of operating solely on “external” data that is passed to it via arguments, it may also operate on the attributes of the object to which it is bound.

7.5.4 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 without the ability to do much else. We want the capability to ask the object to do something useful, or 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 is responsible 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 in the URL. The Zope object responds to the object publisher with a return value, and the object publisher returns the value to your browser.

7.5.5 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 would be members of the same class.

A class is to an object as a set of blueprints is to a house: as many houses can be constructed using the same set of blueprints, many objects can be constructed using the same class. Objects that share a class typically behave identically to one 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 be in the same places, 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 that doesn’t change its essential similarity with the other. It is much the same with instances of a class: if you “visit” two instances of a class, you would 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 exactly the same way.

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

The objects constructed by a class are called instances of the class, or (more often) just instances. For example, the Zope ‘index’ page is an instance of the ‘Page Template’ class. The ‘index’ page has an ‘id’ attribute of ‘index’, while another page may have an ‘id’ attribute of ‘my_page’. However, while they have different attribute values, since they are both instances of the same class, they both behave identically. All the objects that can be administered using the ZMI are instances of a class. Typically, the classes from which these objects are constructed are defined in the add-ons created by Zope developers and community members.

### 7.5.6 Inheritance

It is sometimes desirable for objects to share the same essential behavior, except for small deviations. For example, you may want to create a ContractedEmployee object that has 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. Since classes define behavior, if we want to change an object’s behavior, we almost always need to change its class.

If we base our new “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 idly by and accept all the method definitions of its superclass if they don’t suit its needs: if necessary, the subclass can override the method definitions of its superclass. For instance, we may want 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 with a different implementation. For example, it may always return “Contractor” instead of a job-specific title.

Inheritance is used extensively in Zope objects. For example, the Zope “Image” class inherits its behavior from the Zope “File” class, since images are really just another kind of file, and both classes share many behavior requirements. But the “Image” class adds a bit of behavior that allows it to “render itself inline” by printing its content within HTML tags, instead of causing a file download. It does this by overriding the ‘index_html’ method of the File class.

### 7.5.7 Object Lifetimes

Object instances have a specific lifetime, which is typically controlled by either a programmer or a user of the system in which the objects “live”.

Instances of web-manageable objects in Zope, such as Files, Folders, and Page Templates, span from the time the user creates them until they are deleted. You will often hear these kinds of objects described as persistent objects. These objects are stored in Zope’s object database (the ZODB).

Other Zope object instances have different lifetimes: some object instances 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, which lasts from the moment the object publisher receives the request from a remote browser, until a response
is sent back to that browser, after which it is destroyed automatically. Zope “session data” objects have another well-defined lifetime, which spans from the time a programmer creates one on behalf of the user via 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 to 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.

7.5.8 Summary

Zope is an object-oriented development environment. Understanding Zope fully requires a grasp of the basic concepts of object orientation, including attributes, methods, classes, and inheritance, before setting out on a “for-production” Zope development project.

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,” or the extensive Object FAQ.

7.6 Using the Zope Management Interface

Attention: This document was written for Zope 2.

7.6.1 Introduction

When you log in to Zope, you are presented with the Zope Management Interface (ZMI). The ZMI is a management and configuration environment that allows you to control Zope, manipulate Zope objects, and configure 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 some 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 ZMI. In fact, for the purposes of this chapter, you can safely replace the word “object” with the word “thing” with no ill effects. If you do find something confusing, however, you may want to review the Object Orientation chapter for more detail on objects.

7.6.2 How the Zope Management Interface Relates to Objects

Unlike web server applications like Apache or Microsoft IIS, Zope does not “serve up” HTML files that it finds on your server’s hard drive. Similarly, the objects that Zope creates are not stored in “.html” files on your server. 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 called the “Zope Object DataBase”, or the ZODB. In default configurations, the ZODB creates a file named “Data.fs” in which Zope stores its objects. The ZMI is the primary way by which you interact with Zope objects 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.
7.6.3 ZMI Frames

The ZMI uses three browser frames:

- The left frame is called the *Navigator Frame*, which can be used to expand or collapse a view into the Zope object hierarchy, much like you would expand and collapse a view of files using a file tree widget like the one in Windows Explorer.
- The right frame is called the *Workspace Frame*, which displays a particular view of the object you’re currently managing.
- The top frame is called the *Status Frame*, which displays your user name (when logged in), as well as a drop-down list that performs various actions.

**The Navigator Frame**

In the left-hand, or *Navigator*, 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 rootfolder.

![Fig. 12: The Navigator Frame](image)

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 inside them.

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 root folder is displayed as the current object. The workspace gives you information about the current object and lets you manage it.

A series of tabs is displayed across the top of the screen. The tab that is currently active is highlighted in a lighter color. Each tab takes you to a different view of the current object, and each view lets you perform a different management function on that object.

When you first log into 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.
Fig. 13: The Workspace Frame
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 are used 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’ll 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

The “status frame” at the top of the management interface displays your current login name, 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 you want the management interface to try to use style sheets. Additionally, you can change the default height and width of text-area boxes 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 behave properly with all browsers. If you experience problems logging out using this method, try closing and reopening your browser to log out.

![Fig. 14: The Status Frame](image)

7.6.4 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.

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, as it has no sub-objects or contents). Note 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 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 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. As a result, 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. If our site were called “mysite.example.com”, the new folder’s URL would be “http://mysite.example.com/zoo”.

### 7.6.5 Moving and Renaming Objects

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

To experiment with copy and paste, create a new Folder object 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. Click the Paste button to paste the cut object 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 confirming that bears is no longer visible there.

Copy works similarly to cut, in that, 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, and 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.

Type in the new id value “zoo2” and click OK. Zope ids can consist of letters, numbers, spaces, dashes, underscores, and periods, and they are case-sensitive. Here are some legal Zope ids: index.html, 42, Lucky13, and Snake-Pit.

Now your root folder contains zoo and zoo2 folders. Each of these folders contains a bears folder. This is because when we made a copy of the zoo folder, we 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 and standard_error_message, but it is not recommended to do so unless you have a very good reason.

7.6.6 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 storing objects on the file system 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 an error occurs during the processing of the request. If there is 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.

Most actions 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. This includes undo actions themselves, which can also be undone to restore an object to its state before the undo action.

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.

![Fig. 17: The Undo view](image)

Transactions are named after the Zope action, or “method”, that initiated them. In this case, the initiating method was one named /manage_delObjects, which is the name of the Zope action that deletes Zope objects.

Select the first transaction labeled /manage_delObjects, and click the _Undo_ button at the bottom of the form. Doing so instructs Zope to undo the last transaction. You can verify that the task has been completed by visiting the root folder to confirm that the _zoo_ folder has returned. If you use the “Back” button to revisit the root folder, you may need to refresh your browser to see the proper results. To see the effect in the _Navigator_ pane, click the “Refresh” link within the pane.

You may “undo an undo” action, or “redo” the action, and you can undo and redo actions as many times as you like. When you perform a “redo”, Zope inserts a transaction into the undo log describing the redo action.

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 affected the current object and its sub-objects.
7.6.7 Undo Details and Gotchas

You cannot undo a transaction upon which a later transaction depends. For example, if you paste an object into a folder, and then delete an object in the same folder, pasting the first object cannot be undone, as both transactions affect the contents of a single object: the folder. 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 into a relational database server, such as Oracle or MySQL (as discussed in the chapter entitled “Relational Database Connectivity”), changes to data stored there cannot be undone.

7.6.8 Reviewing Change History

The Undo tab will provide you with enough information to know that a change has occurred. However, it will not tell you much about the effect of the transaction on the objects that were changed during the transaction.

7.6.9 Using Object Properties

Properties are ways of associating information with many objects in Zope, including folders. For example, many Zope content objects have a content type property, and others contain metadata about the object, such as its author, title, or status.

Properties can provide more complex data than strings, such as numbers, lists, and 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.

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, which has the value ‘Zope’. You may change any predefined property by changing its value in the Value box, and then clicking Save Changes. You may add additional properties to an object by entering a name, value, and type into the bottom-most field in the Properties view.

Zope supports a number of property types and 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 a sequence of characters of arbitrary length. 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 to be at least 32 bits long.

**long** A long is 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 a 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, in that it is used to render an HTML single selection input widget.

**multiple selection** A multiple selection property is special, in that it is used to render an HTML multiple selection form input widget.

Properties are very useful tools for tagging your Zope objects with bits of metadata. Properties are supported by most Zope objects and are often referenced by your application logic for purposes of data display.

### 7.6.10 Logging Out

You may choose Logout from the Status Frame drop-down 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, you may actually need to click on the “OK” button with an **incorrect** user name and password in the authentication dialog in order to effectively log out of the ZMI. If you do not do so, you may find 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.

### 7.7 Using Basic Zope Objects

**Attention:** This document was written for Zope 2.

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

#### 7.7.1 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 by 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 like 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, such as information 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: Zope Page Templates (ZPT) and Document Templates (DTML). In the first part of the book we will only cover page templates and later on expand on document templates. If you already know HTML, page templates are easier to work with and more limited in their options. For some of the more advanced tasks DTML can be a better option as explained later on.

**Logic** Scripting business logic in Zope is done using Python. “Logic” is any kind of programming that does not involve presentation, but rather involves the carrying out of tasks such as changing objects, sending messages, testing conditions, and responding to events.

Zope also has other kinds of objects that fit into none of these categories, which are explored further in the chapter entitled Zope Services. You may also install “third party” Zope objects, defined in Python packages, to expand Zope’s capabilities. You can browse a list of packages specifically aimed at Zope at the Python Package Index.

### 7.7.2 Content Objects: Folders, Files, and Images

**Folders**

You’ve already met one of the fundamental Zope objects: the Folder. Folders are the basic building blocks of Zope. The purpose of a folder is simple: a Folder’s only job in life is to 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 like Microsoft Windows Explorer.

**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, audio files, “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 different categories 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 a File object in your Zope instance, visit the root folder in the ZMI and choose 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 trigger your browser to display a dialog box that allows 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.

![Fig. 19: Adding a PDF File Object](image)

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 you select 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.

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 object editing interface. To change the content type of a File object, type the new content type into the Content Type field and click the Save Changes button.

You can change the contents of an existing File object by selecting a new file from your local file system in the File Data form element and clicking Upload.

Editing Text File Contents

If your File holds only text and is smaller than 64 kilobytes, Zope will allow you to edit its contents in a textarea within the Edit view of the ZMI. 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 in a File object’s management screen.

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, you can view that file in your web browser via the URL http://localhost:8080/Reader.pdf. Depending on the type of file and your web browser’s configuration, your web browser may choose to display or download the file.

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, except that they include extra behavior for managing graphic content, such as an image’s width and height attributes.

Image objects use the same management interface as File objects. Everything in the previous section about using file objects also applies to images. In addition, Image objects display a preview of their images once they have been uploaded to Zope.

7.7.3 Presentation Objects: Zope Page Templates

Zope encourages you to keep your presentation and logic separate by providing different objects that are intended to be used expressly 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).

Zope has one “presentation” facility: Zope Page Templates (ZPT). Zope Page Templates are objects that 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 that define the dynamic behavior for that page.
ZPT has characteristics of a “server-side” scripting language, like SSI, PHP or JSP. This means that 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.

Zope also has an older version of a presentation facility included, which is called Document Template Markup Language or short DTML.

**ZPT vs. DTML: Same Purpose, Different Approach**

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 that 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, and 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” workflow: 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:

- the designer mocks up a new page and the programmer re-embeds the dynamic tags “from scratch”, or
- the designer hand-edits the HTML, working around the dynamic tags, or
- the programmer does the presentation himself.
Clearly, none of these options are desirable, 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 it failed to allow programmers to work effectively together with non-technical 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 non-technical designers.

DTML is still fully supported in Zope. If you are familiar with PHP it might fit your mind better then ZPT. For some of the advanced topics covered later in the book, like relation database integration or more uncommon tasks like dynamic generation of non-xml files, DTML can be easier to work with.

**Zope Page Templates**

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

**Creating a Page Template**

Create a Folder with the ‘id’ Sales in the root folder, and give it any title you like. 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 whose 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 ZMI. When you click on either one of those items, you are taken to the Edit view of the page template, which displays a textarea in which you can edit the template. Click on the “SalesPage” template. You will see something like the following screen:

Replace the original, boilerplate content included in the page template with the following HTML:

```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 ‘upload_pt.html’. Populate it with the following content:

```html
<html>
<body>
  <h1>This is my second page template!</h1>
</body>
</html>
```
While in the Sales folder, choose 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, and your browser will display a file selection dialog. Choose the ‘upload_pt.html’ file, type in an ‘id’ of “upload_pt” for the new Page Template, and click Add and Edit. After uploading your file, you will be taken back to the Edit form of your new 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:

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

### 7.7.4 Logic Objects: Script (Python) Objects and External Methods

“Logic” objects in Zope are objects that typically perform some sort of “heavy lifting” or “number crunching” in support of presentation objects. When they are executed, they need not return HTML or any other sort of structured presentation text. Instead, they might 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 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 meaningfully in a browser. 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**. These stock logic 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](https://docs.python.org/3/) in order to understand the syntax and semantics of the example Script (Python) objects shown throughout this chapter and throughout this book. And don’t panic: Python is very easy to learn and understand.

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” — “{” and “}” — to express a block of code, Python determines code blocks by examining the indentation of 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 objects, make sure to check your Script’s indentation.

### Script (Python) Objects

Script (Python) objects are one type of logic object. Note that the tortuous 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 defined set of restricted Python modules, and that they cannot directly access files on your file system. This is a security feature, as it allows site administrators to safely delegate the ability to create logic in Python to less knowledgeable users. The security restrictions aren’t stringent enough to prevent malicious users from damaging the system. They are only meant as a safety belt beginners. For more information about Zope’s security features, see [Users and Security](https://zope.readthedocs.io/en/latest/security.html).
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, whose 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 ZMI: when you click on either of these items, you are taken to the Edit view of the Script (Python), which gives you a textarea in which you can edit the template. Click on the ‘SalesScript’ icon. You will see something like the following:

Fig. 24: Default Script Content

In the Parameter List form element, type ‘name="Chris"’.

Replace the original content that comes in the “body” (the big TEXTAREA below the ‘Last Modified’ line) of the Script (Python) object with the following text:

```
return 'Hello, %s from the SalesScript script' % name
```

Then click Save Changes at the bottom of the edit form. You can now execute, or test, your Script (Python) object.

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:
In the Value box next to the ‘name’ parameter, enter your name, and 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 and including a “query string”. In our case, visiting the URL ‘http://localhost:8080/Sales/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) object is much like uploading the body of a 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
```

(continues on next page)
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 “auto-magically” turned into bindings and parameters for the Script (Python) object.

**External Methods**

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

- **External Methods** are not editable using the Zope Management Interface. Instead, their “modules” need to be created on the file system of your Zope server in a special subdirectory of your Zope directory named ‘Extensions’.

- Because **External Methods** 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, External Methods can import and execute essentially arbitrary Python code and access files on your Zope server’s file system.

- **External Methods** do not support the concept of “bindings” (which we have not discussed much yet, 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) object 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) object cannot write to files on your server’s filesystem that an External Method may.

**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, its output is 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:

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:

```python
Hello, Fred from the Sales external method
```
Astute readers will note that the ‘id’ provided by the output is *not* the ‘id’ of the External Method (‘SalesEM’), but 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 (Python) object “knows about” its context and can display information about the context without much fuss.

**Creating and Editing an External Method File**

Minimize the browser you’re using to access the ZMI. In your Zope’s INSTANCE_HOME (the place where your Zope instance lives; see the Installation chapter for details), locate the subfolder named ‘Extensions’. Navigate into this folder and create a text file with the name ‘SalesEM.py’.

Within this file, save the following content:

```python
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 the ZMI that “refers to” the function in the file that you just created. Bring back your browser window and visit the ZMI. 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”.

---

Fig. 26: Testing an External Method
Then click Add at the bottom of the Add form.

**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.

### 7.7.5 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 that 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 (Python) objects to construct an application to perform this task.

For this example, you will need two Page Templates with the ‘ids’ *interestRateForm* and *interestRateDisplay*, respectively, to collect and display information from the user. You will also need a Script (Python) object 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 with the ‘id’ “Interest”. You will create all of the objects that 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
<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 ">
</form>
</body>
</html>
```
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”:

```python
***
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 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
<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: context.calculateCompoundingInterest(principal,
        interest_rate, 
        periods, 
        years)" >1.00</span>
</b></span>
</body>
</html>
```

Dealing With Errors

As in any programming venue, you will need to deal with errors. Nobody’s perfect! You may have already encountered some errors as you 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”. Thus, 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 following text:
An error was encountered 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. You can view the full error by visiting the error_log object and clicking the top-most error log entry, which will be named KeyError: years in the Log tab. The error log entry contains information about the error, including the time, the user who received the error, the URL that caused the error to happen, the exception type, the exception value, and a “Traceback”, which typically gives you enough technical 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 the ‘request’ “namespace”. We’ll cover Page Template namespaces in a succeeding chapter.

### 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 then click Calculate. This will cause `interestRateForm` to submit the collect information to `interestRateDisplay`, which calls the Script (Python) object named `calculateCompoundingInterest`. The display method uses the value returned by the script in the resulting display. You will see the following result:

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 trouble, try to troubleshoot the application by using the tips in the section “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 as concisely and clearly as possible. It is likely that someone there will be able to help you, and it is polite to subscribe to the Zope mailing list 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) mailing list.

### 7.8 Acquisition

**Attention:** This document was written for Zope 2.

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

Acquisition’s flavor permeates Zope and can be used almost everywhere within Zope: 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.
Over the years Acquisition has been proven to be a very powerful but often too complex technology to use. While it is predictable in simple interactions, it gets increasingly complicated to understand its behavior in most real-world-sized projects.

In order to understand Zope, you will still need an understanding of Acquisition today. Basing your application logic on it is highly discouraged, though.

### 7.8.1 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 that define 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 (parentheses) causes that class to inherit behavior from the classes named in the parentheses (you can play along at home if you like, using the Python interpreter):

```python
>>> 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"
...
```

(continues on next page)
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. The case of the Sub class above 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. If it doesn’t find it there, it will search in the SuperB class. Python performs this search of the base classes using an order derived from the order of declaration. Note that for complex cases (e.g., where the same method is defined in several ancestors of base classes), the lookup order is too complicated to explain within the scope of this book. Please see the online Python documentation for the “method resolution order”, http://www.python.org/download/releases/2.3/mro/

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

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

### 7.8.2 Acquisition is about Containment

The concept behind acquisition is simple:

- Objects are situated inside other objects, and these objects act as their “containers”. For example, the container of a Page Template named “apage” inside a Folder “afolder” is the “afolder” 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 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, then the acquisition hierarchy is searched.

7.8.3 Say What?

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

Place a Page Template named ‘acquisition_test’ in your Zope root folder. Give it the following body:

```html
<html>
<body>
<p>
I am being called from within the
<span tal:replace="context/title" />
Folder!
</p>
</body>
</html>
```

Save it, and then use the Page Template “View” tab to see the result of the template in your Workspace frame. You will see something not unlike the following:

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

The ‘title’ of the Zope root folder is ‘Zope’, so this makes sense. Now create a Folder inside your Zope root folder named ‘AcquisitionTestFolder’ and a title of “TheAcquisitionTest”. We’re going to invoke the ‘acquisition_test’ page 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 TheAcquisitionTest Folder!
```

Note that even though an object named ‘acquisition_test’ does not “live” inside the AcquisitionTestFolder folder, Zope found the page and displayed a result anyway! Not only did Zope display a result, instead of inserting the ‘title’ of the Zope root folder, it inserted the ‘title’ 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 adding an ‘acquisition_test’ page).

7.8.4 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’ page.

Not only do objects acquire services, but 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 in order to make services available to other objects.
7.8.5 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 Page Template, “HappySong”, in the root folder, and 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 page is found in the root folder, unless one of the other folders “Users”, “Barney” or “Songs” also have a page named “HappySong”, in which case that page is used instead. The HappySong page 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 at all possible.

In the example above, for instance, in order to find and publish the “HappySong” template at the end of the URL, acquisition searches the containment hierarchy of the “Songs” folder first. Because “Songs” is contained within “Barney”, and “Barney” within “Users”, the containment hierarchy for “Songs” consists of each folder “up” from “Users” to the root.

Once the “HappySongs” template is found, there are two hierarchies of interest:

- Because “HappySongs” is located directly within the root, its containment hierarchy consists of only itself and the root.
- Because “HappySongs” was found by traversing first through the “Users”, “Barney”, and “Songs” folders, its context hierarchy includes those objects.

Acquisition searches the context hierarchy only after failing to find the named object in the containment hierarchy.

As with understanding Python’s concept of multiple inheritance, explaining the exact strategy used to order that search is not within the scope of this book.

7.8.6 Summary

Acquisition allows behavior to be distributed hierarchically throughout the system. When you add a new object to Zope, you don’t need to specify all of its behavior, only the part of its behavior that is unique to that object. 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 that 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.
7.9 Basic Zope Scripting

Attention: This document was written for Zope 2.

So far, you’ve learned about some basic Zope objects and how to manage them through the Zope Management Interface. This chapter shows you how to manage Zope objects programmatically.

7.9.1 Calling Methods From the Web

Since Zope is a web application server, the easiest way to communicate with Zope is through a web browser. Any URL your browser requests from the server is mapped to both an object and a method. The method is executed on the object, and a response is sent to your browser.

As you might already know, visiting the URL

http://localhost:8080/

returns the Zope Quick Start page. In this case, we only specify an object – the root object – but no method. This just works because there is a default method defined for Folders: index_html. Visiting the URL:

http://localhost:8080/index_html

returns (almost) exactly the same page.

You can also specify the root object as:

http://localhost:8080/manage_main

but in this case the manage_main method is called, and the workspace frame displays the root content of your Zope site, without the navigator frame.

The same method can be called on other objects: when you visit the URL:

http://localhost:8080/Control_Panel/manage_main

the manage_main method is called on the Control Panel object.

Sometimes a query string is added to the URL, e.g.:

http://localhost:8080/manage_main?skey=meta_type

The query string is used for passing arguments to the method. In this case, the argument skey specifies the sort key with the value meta_type. Based on this argument, the manage_main method returns a modified version of the basic page: the sub-objects are sorted by Type, not by Name as they are without that query string.

While the manage_main method is defined in the class of the object, index_html is (by default) a DTML Method object in the root folder that can be modified through the web. index_html itself is a presentation object, but when called on a folder, it behaves as a method that returns the default view of the folder.

Method Objects and Context Objects

When you call a method, you usually want to single out some object that is central to the method’s task, either because that object provides information that the method needs, or because the method will modify that object. In object-oriented terms, we want to call the method on 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 object can be used as a method for (potentially) many other objects, without its being defined by 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 an object 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 method gives it a context in which to execute. Or, to put it another way: the context is the environment in which the method executes, from which the method may get information that it needs in order to do its job.

Another way to understand the context of a method is to think of the method as a function in a procedural programming language, and its context as an implicit argument to that function.

While the Zope way to call methods in the context of objects works differently than the normal object-oriented way to call class-defined methods on objects, they are used the same way, and it is simpler to say that you are calling the method on the object.

There are two general ways to call methods on objects: by visiting an URL, and by calling the method from another method.

**URL Traversal and Acquisition**

The concept of calling methods in the context of objects is a powerful feature that enables you to apply logic to objects, like documents or folders, without having to embed any actual code within the object.

For example, suppose you have a collection of objects and methods, as shown in the figure below.

![Fig. 28: A collection of objects and methods](image)

To call the feed method on the hippo object, you would visit the URL:
To call the `feed` method on the `kangarooMouse` object you would visit the URL:

```
Zoo/SmallAnimals/kangarooMouse/feed
```

These URLs place the `feed` method in the context of the `hippo` and `kangarooMouse` objects, respectively. 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` method 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` method 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` method 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.

Note that *Script (Python)* and *Page Template* objects are always method objects. You can’t call another method in the context of one of them. Given `wash` is such a method object, visiting the URL

```
Zoo/LargeAnimals/hippo/wash/feed
```

would also call the `wash` method on the `hippo` object. Instead of traversing to `feed`, everything after the method `wash` is cut off of the URL and stored in the variable `traverse_subpath`.

### The Special Folder Object `index_html`

As already mentioned at the beginning of this chapter, Zope uses the default method if no other method is specified. The default method for Folders is `index_html`, which does not necessarily need to be a method itself. If it isn’t a callable, the default method of the object `index_html` is called on `index_html`. 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 and still gain the same effect.

For example, if you create an `index_html` object in your `Zoo` Folder, and view the folder by clicking the View tab or by visiting the URL:

```
http://localhost:8080/Zoo/
```

Zope will call the `index_html` object in the `Zoo` folder and display its results. You could instead use the more explicit URL:

```
http://localhost:8080/Zoo/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 that 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 may 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 display the root Folder’s `index_html`. This is acquisition at work.

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 Page Template, a Script (Python) object, a DTML Method or any other Zope object that is URL-accessible and that returns browser-renderable content. The content is typically HTML, but Zope doesn’t care. You can return XML, or text, or anything you like.

### 7.9.2 Using Python-based Scripts

Now let us take a look at a basic method object: `Script (Python)`.

**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 website has lots of Python documentation including a tutorial by Python’s creator, Guido van Rossum.

For people who have already some programming experience, Dive Into Python is a great online resource to learn python.

Python comes with a rich set of modules and packages. You can find out more about the Python standard library at the python.org website.

**Creating Python-based Scripts**

To create a Python-based Script, select `Script (Python)` from the Add drop-down 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 in the body of the script, type:

```
return "Hello %s." % name
```

Our script is now equivalent to the following function definition in standard Python syntax:

```
def hello(name="World"):
    return "Hello %s." % name
```
The script should return a result similar to the following image:

Fig. 29: Script editing view

You can now test the script by going to the Test tab, as shown in the following figure.

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 clicking the Run Script button. Zope should now say “hello” to you.

Since scripts are called on Zope objects, you can get access to Zope objects via the context variable. For example, this script returns the number of objects contained by a given Zope object:

```python
## Script (Python) "numberOfObjects"
##
return len( context.objectIds() )
```

Note that the lines at the top starting with a double hash (##) are generated by Zope when you view the script outside the Edit tab of the ZMI, e.g., by clicking the view or download link at the bottom of the Edit tab. We’ll use this format for our examples.

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. 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':
This script loops through an object’s sub-objects, and returns all the sub-objects that have a:

status

property with a given value.

**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

---

7.9. Basic Zope Scripting
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:

```python
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. Due to security restrictions, you cannot do regular expression processing within Python-based Scripts. If you really need regular expressions, you can easily use them in External Methods, described in a subsequent chapter. However, in a Script (Python) object, you do have access to string methods.

Suppose you want to change all the occurrences of a given word in a text file. Here is a script, `replaceWord`, that accepts two arguments: `word` and `replacement`. This will change all the occurrences of a given word in a File:

```python
## Script (Python) "replaceWord"
##parameters=word, replacement
##
## """ Replaces all the occurrences of a word with a replacement word in the source text of a text file. Call this script on a text file to use it.
##
## Note: you will need permission to edit the file in order to call this script on the *File* object. This script assumes that the context is a *File* object, which provides 'data', 'title', 'content_type' and the manage_edit() method.
## """

text = context.data
text = text.replace(word, replacement)
context.manage_edit(context.title, context.content_type, filedata=text)
```

You can call this script from the web on a text *File* in order to change the text. For example, the URL:

```
Swamp/replaceWord?word=Alligator&replacement=Crocodile
```

would call the `replaceWord` script on the text *File* named:

```
Swamp
```

and would replace all occurrences of the word:

```
Alligator
```

with:

```
Crocodile
```

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 within 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.
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 the service does not always 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:

```python
## 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 controlling 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’d 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 file system. 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, and 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 to avoid accidental long execution times.

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:

```python
if type(foo) == type([]):
    return "foo is a list"
```

... to check if `foo` is a list, you would instead use the `same_type` function:

```python
if same_type(foo, []):
    return "foo is a list"
```
**7.9.3 Calling ZPT from Scripts**

Often, you would want to call a *Page Template* 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 *Page Template* and call it from the Script. Assume we have this Page Template with the id `hello_world_pt`:

```html
<p>Hello <span tal:replace="options/name | default">World</span>!</p>
```

You will learn more about *Page Templates* in the next chapter. For now, just understand that this *Page Template* generates an HTML page based on the value `name`. Calling this template from a Script and returning the result could be done with the following line:

```python
return context.hello_world_pt(name="John Doe")
```

The `name` parameter to the Page Template ends up in the:

```python
options/name
```

path expression. So the returned HTML will be:

```html
<p>Hello John Doe!</p>
```

Note that:

```python
context.hello_world.pt
```

works because there is no dot in the id of the template. In Python, dots are used to separate ids. This is the reason why Zope often uses ids like `index_html` instead of the more common `index.html` and why this example uses `hello_world_pt` instead of `hello_world.pt`.

However, if desired, you can use dots within object ids. Using `getattr` to access the dotted id, the modified line would look like this:

```python
return getattr(context, 'hello_world.pt')(name="John Doe")
```

**7.9.4 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 name 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.

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:

```python
## 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’ve simply hard-coded some numbers.

You could call this script from ZPT like this:

```
<p tal:repeat="diet context/compute_diets">
  This animal needs
  <span tal:replace="diet/fat" />kg fat,
  <span tal:replace="diet/protein" />kg protein, and
  <span tal:replace="diet/carbohydrate" />kg carbohydrates.
</p>
```

More on ZPT in the next chapter.

### 7.9.5 The Zope API

One of the main reasons to script in Zope is to get convenient access to the Zope Application Programmer Interface (API). 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.

Suppose you would like a script that takes a file you upload from a form, and creates a Zope File object in a Folder. To do this, you’d need to know a number of Zope API actions. It’s easy enough to read files in Python, but once you have the file, you must know which actions to call in order 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, including 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`,...
which means that Folder objects have all the methods 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 containing the ids of the object’s containment hierarchy:

```
path_list = context.getPhysicalPath()
pth_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)
```

### 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:
Change properties of an object

The object has to support properties, and the property must exist:

```python
values = {'pattern': 'spotted'}
context.manage_changeProperties(values)
```

Traverse to an object and add a new property

We get an object by its absolute path, add a property `weight` and set it to some value. Again, the object must support properties in order for this to work:

```python
path = "/Zoo/LargeAnimals/hippo"
hippo_obj = context.restrictedTraverse(path)
hippo_obj.manage_addProperty('weight', 500, 'int')
```

7.10 Using Zope Page Templates

**Attention:** This document was written for Zope 2.

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 basic features of Page Templates, including how you can use them in your website to create dynamic web pages easily. The next chapter walks you through a “hands on” example showing how to build a Zope web application using scripts and Page Templates. 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.

A Page Template is like a model of the pages that it will generate. In particular, it is parseable by most HTML tools.
7.10.1 HTML Page Templates

Page Templates can operate in two modes: HTML Mode and XML Mode. Later in this chapter we will show you how to use the XML Mode, but in most cases we want to use the HTML Mode which is also the default mode. For the HTML Mode the Content-Type has to be set to ‘text/html’.

HTML isn’t XML-conform and can’t be extended by a template language. So while rendered HTML Page Templates should return valid HTML, their source code isn’t valid HTML or XML. But the Template Attribute Language (TAL) does a good job in hiding itself in HTML tags, so most HTML tools will be able to parse the source of HTML Page Templates and just ignore the TAL attributes.

As you might already know, XHTML is a XML-conform reformulation of HTML and widely used in our days. Nevertheless, generating HTML and XHTML with Page Templates works exactly the same way. While the HTML Mode doesn’t enforce well-formed XML, it’s absolutely fine to use this mode also for XHTML.

How Page Templates Work

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

```html
<h1 tal:content="context/title">Sample Page Title</h1>
```

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 ‘h1’ tag, and the value ‘context/title’ is an expression providing the text to insert into the tag. Given the text specified by ‘context/title’ resolves to “Susan Jones Home Page”, the generated HTML snippet looks like this:

```html
<h1>Susan Jones Home Page</h1>
```

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 headline example is perfectly parseable HTML, and shows up in their editor looking like a headline 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 headline text will act as a placeholder for the dynamic headline 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 “Sample Page Title” with whatever ‘context/title’ resolves to. In this case, ‘context/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

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. 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 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
<html>
<body>
<p>
  This <b tal:content="template/title">is 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 you get an error message, check to make sure you typed the example correctly and save it again.

Click on the Test tab. You should see a page with “This is a Simple Page” at the top. Notice that the title is bold. This is because the ‘tal:content’ statement just replaces the content of the bold 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.

You can find two options on the Edit tab we will not touch for now: The Content-Type field allows you to specify the content type of your page. Changing that value switches the Page Template into XML Mode, discussed later in this chapter. The Expand macros with editing control is explained in the “Macros” section of this chapter.

**TALES Expressions**

The expression “template/title” in your simple Page Template is a path expression. This is 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 Appendix C, Zope Page Templates Reference.

**Path Expressions**

The ‘template/title’ path expression fetches the title attribute of the template. Here are some other common path expressions:

- ‘context/objectValues’: A list of the sub-objects of the folder on which the template is called.
- ‘request/URL’: The URL of the current web request.
- ‘user/getUserName’: The authenticated user’s login name.

From the last chapter you should already be familiar with the context variable that is also available in Python-based Scripts and the attribute ‘objectValues’ that specifies an API method. The other two examples are just to show you the pattern. You will learn more about them later in the book.

To see what these examples return, just copy the following lines into a Page Template and select the Test tab. You’ll notice that ‘context/objectValues’ returns a list that needs further treatment to be useful. We’ll come back to that later in this chapter.
Every *path expression* starts with a variable name. The available variable names refer either to objects like *context*, *request* or *user* that are bound to every *Page Template* by default or variables defined within the *Page Template* using *TAL*. Note that *here* is an old alias of *context* and still used in many places.

The small set of built-in variables such as *request* and *user* is described in the chapter entitled *Advanced Page Templates*. You will also learn how to define your own variables in that chapter.

If the variable itself returns the value you want, you are done. Otherwise, you add a slash (‘/’) and the name of a sub-object or attribute. You may need to work your way through several sub-objects to get to the value you’re looking for.

### Python Expressions

A good rule of thumb is that if you need Python to express your logic, you better factor out the code into a script. But Zope is a good tool for prototyping and sometimes it would be overkill to write a script for one line of code. And looking at existing products you will see quite often ‘Python expressions’, so it’s better to know them.

Recall the first example of this chapter:

```
<h1 tal:content="context/title">Sample Page Title</h1>
```

Let’s try to rewrite it using a *Python expression*:

```
<h1 tal:content="python: context.title">Sample Page Title</h1>
```

While *path expressions* are the default, we need a prefix to indicate other expression types. This expression with the prefix ‘python:’ does (at least here) the same as the *path expression* above. *Path expressions* try different ways to access ‘title’, so in general they are more flexible, but less explicit.

There are some simple things you can’t do with *path expressions*. The most common are comparing values like in:

```
"python: variable1 == variable2"
```

... or passing arguments to methods, e.g.:

```
"python: context.objectValues(['Folder'])"
```

### TAL Attributes

*Page Templates* are example pages or snippets. *TAL* statements define how to convert them dynamically. Depending on the used *TAL* attribute they substitute example content or attributes by dynamic values, or remove or repeat example elements depending on dynamic values.

### Inserting Text

In your “simple_page” template, you used the ‘tal:content’ statement on a *bold* tag. When you tested it, Zope replaced the content of the HTML *bold* element with the title of the template.

This is easy as long as we want to replace the complete content of an HTML element. But what if we want to replace only some words within an element?
In order to place dynamic text inside of other text, you typically use ‘tal:replace’ on an additional ‘span’ tag. For example, add the following lines to your example:

```xml
<p>The URL is <span tal:replace="request/URL">
http://www.example.com</span>.</p>
```

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:

```xml
The URL is http://localhost:8080/template_test/simple_page.
```

If you look at the source code of the rendered version, the span tags are removed.

To see the difference between ‘tal:replace’ and ‘tal:content’, create a page template and include the following in the body:

```xml
<b tal:content="template/title"></b>
<b tal:content="request/URL"></b>
<b tal:content="user/getUserName"></b>
<b tal:replace="template/title"></b>
<b tal:replace="request/URL"></b>
<b tal:replace="user/getUserName"></b>
```

There are two other ways to add elements that are only needed for TAL attributes and that are removed again in the rendered version:

```xml
<p>The URL is <span tal:content="request/URL" tal:omit-tag="">
http://www.example.com</span>.</p>
```

... which is more useful in other situations and will be discussed there and:

```xml
<p>The URL is <tal:span tal:content="request/URL">
http://www.example.com</tal:span>.</p>
```

While you can get really far by using HTML elements and ‘tal:replace’ or ‘tal:omit-tag’, some people prefer to use TAL elements if the elements are only used to add TAL attributes. TAL is an attribute language and doesn’t define any elements like ‘tal:span’, but it uses a complete XML namespace and allows to use any element name you like. They are silently removed while the Page Template is rendered.

This is useful for using speaking names like ‘tal:loop’, ‘tal:case’ or ‘tal:span’ and to insert additional elements where HTML doesn’t allow elements like ‘span’ or ‘div’. And if her browser or editor also ignores these tags, the designer will have less trouble with TAL elements than with additional HTML elements.

**Repeating Structures**

Let’s start with a simple three-liner:

```xml
<p tal:repeat="number python: range(4)" tal:content="number">
  999
</p>
```
‘number’ is our repeat variable and ‘range(4)’ is a Python expression that returns the list ‘[0, 1, 2, 3]’. If this code is rendered, the ‘repeat’ statement repeats the paragraph element for each value of the sequence, replacing the variable ‘number’ by the current sequence value. So the rendered page will not show the example number ‘999’, but 4 paragraph elements containing the numbers of our list.

In most cases we want to iterate over more complex sequences. Our next example shows how to use a sequence of (references to) objects. The ‘simple_page’ template could be improved by adding an item list, 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 row for each object, and columns for the id, meta-type and title. Add these lines to the bottom of your example template:

```html
<table border="1" width="100%">
  <tr>
    <th>Id</th>
    <th>Meta-Type</th>
    <th>Title</th>
  </tr>
  <tr tal:repeat="item context/objectValues">
    <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 context’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. See Advanced Page Templates.

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:

```html
<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.

To allow you to see the effect, we first have to modify that example a bit, showing only Folder objects in the context folder. Because we can’t specify parameters using path expressions like ‘context/objectValues’, we first convert it into the Python expression ‘context.objectValues()’ and then add the argument that tells the ‘objectValues’ method to return only sub-folders:

```html
<tr tal:repeat="item python: context.objectValues(['Folder'])">  
</tr>
```

If you did not add any sub-folders to the template_test folder so far, you will notice that using the Test tab the table header is still shown even if we have no table body. To avoid this we add a ‘tal:condition’ statement in the table tag. The complete table now looks like this:

```html
<table tal:condition="python: context.objectValues(['Folder'])" border="1" width="100%">
  <tr>
    <th>Id</th>
    <th>Meta-Type</th>
    <th>Title</th>
  </tr>
  <tr tal:repeat="item python: context.objectValues(['Folder'])">
    <td tal:content="item/getId">Id</td>
    <td tal:content="item/meta_type">Meta-Type</td>
    <td tal:content="item/title">Title</td>
  </tr>
</table>
```

If the list of sub-folders is an empty list, the condition is false and the entire table is omitted. You can verify this by using the Test tab again.

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:

<table>
<thead>
<tr>
<th>Id</th>
<th>Meta-Type</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Folder</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Folder</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Folder</td>
<td></td>
</tr>
</tbody>
</table>

### Changing Attributes

Most, if not all, of the objects listed by your template have an icon attribute 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:

```html
<td><img src="file_icon.gif" tal:attributes="src item/icon" />
  <span tal:replace="item/meta_type">Meta-Type</span></td>
```

The ‘tal:attributes’ statement replaces the ‘src’ attribute of the ‘img’ tag with the value of ‘item/icon’. The ‘src’ attribute in the template (whose value is “file_icon.gif”) 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 XML with *Page Templates* is almost exactly like creating HTML. You switch to **XML Mode** by setting the `content-type` field to `text/xml` or whatever the content-type for your XML should be.

In **XML Mode** no “loose” markup is allowed. 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:

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

To browse the source of an XML template you go to ‘source.xml’ rather than ‘source.html’.

**Debugging and Testing**

**Zope** helps you find and correct problems in your *Page Templates*. Zope notices problems 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 and 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:

```xml
<!-- 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*.

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

```plaintext
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:

```plaintext
Error Type: KeyError
Error Value: 'unicorn'
```

This error message tells you that it cannot find the *unicorn* variable. 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 place where the error occurred and the environment:

```plaintext
URL: /sandbox/demo
Line 1, Column 14
Expression: standard:'context/unicorn'
Names: {'container': <Folder instance at 019AC4D0>,
```

(continues on next page)
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 ‘context’ 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 root on which you called the template hasn’t such a property.

7.10.3 Macros

So far, you’ve seen how Page Templates can be used to add dynamic behavior to individual web pages. Another feature of page templates is the ability to reuse look and feel elements across many pages.

For example, with Page Templates, you can have a site that has a standard look and feel. No matter what the “content” of a page, it will have a standard header, side-bar, footer, and/or other page elements. This is a very common requirement for websites.

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 2009, <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, ending with the closing ‘p’ tag).

Macros defined in a Page Template are stored in the template’s macros attribute. You can use macros from other Page Templates 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 the 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 2009, <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:

```xml
<p metal:use-macro="python:context.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. An example of the body of a “getMacro” Script (Python) is as follows:

```
return container.ptMacros.macros['amacroname']
```

You can use the ‘default’ variable with the ‘metal:use-macro’ statement:

```xml
<p metal:use-macro="default">
  This content remains - no macro is used
</p>
```

The result is the same as using *default* with ‘tal:content’ and ‘tal:replace’. The “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:

```xml
<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.
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:

```html
<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:

```html
<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:

```html
<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:

```html
    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:

```html
<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:

```html
<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:

```html
<ul metal:define-macro="links">
  tal:repeat="link context/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
<html metal:define-macro="page">
<head>
<title tal:content="context/title">The title</title>
</head>
<body>
<h1 metal:define-slot="headline"
tal:content="context/title">title</h1>
<p metal:define-slot="body">
This is the body.
</p>
<span metal:define-slot="footer">
<p>Copyright 2009 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
<html metal:use-macro="container/master.html/macros/page">
<h1 metal:fill-slot="headline">
Press Release:
<span tal:replace="context/getHeadline">Headline</span>
</h1>
<p metal:fill-slot="body"
tal:content="context/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.

7.10.4 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 ships with several content components: ZSQL Methods, Files, and Images. 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 workflow, skins and content objects. The CMF makes a lot of use of Page Templates.

7.11 Creating Basic Zope Applications

Attention: This document was written for Zope 2.

This chapter will take you, step by step, through building a basic web application in Zope. As we go through the chapter, we will examine some of Zope’s main concepts at work. Using Zope Folder, Script (Python), and Page Template objects, we’ll create a simple website for an imaginary zoo: the “Zope Zoo”, of course!

We will develop the website as a Zope “instance-space” application. A discussion of instance space is at the end of this chapter, but for now it is enough to know that instance-space applications are the easiest and fastest kind to build, because we can do everything in our favorite web browser.

7.11.1 Goals for the Zope Zoo Web Site

As with any project, we first need to clarify our goals for the Zope Zoo application. The application’s primary goal is to create a website for the world-renowned Zope Zoo. Furthermore, we want to make the website easy to use and manage. Here are some things we’ll do:

1. Enable web users to navigate the site easily, as if they were moving around a real zoo.
2. Keep all our shared web layout tools, like a Cascading Style Sheet (CSS), in a single, easy-to-manage location.
3. Design the website so that future site-wide changes are quick and easy to implement.
4. Take advantage of Zope to create a dynamic website in which web pages build themselves “on the fly” when requested so that they are always up to date.
5. Provide a simple file library of various documents that describe the animals.

7.11.2 Beginning with a Folder

Zope Folder objects provide natural containers and organizers for web applications. A good way to start building an application is to create a new Folder to hold all the objects and subfolders related to the application.
Consider, for example, a Zope folder named *Invoices* to hold an application for managing invoices through the Web. The *Invoices* folder could contain both the logic objects—or “methods”—which allow you to add and edit invoices, as well as the actual data of the invoices. The *Invoices* folder thus becomes a small Zope application.

We begin building our Zope Zoo website application by creating a *Zope Folder* object to hold it all together in one place.

### Step 1: Create ZopeZoo Folder

If you haven’t already, start your Zope installation and log into the Zope Management Interface (ZMI) using your favorite browser. (If you are not familiar with the ZMI, refer to the *Installing and Starting Zope* chapter.)

1. Navigate to Zope’s top-level *root* folder.
2. Use the *Add* list to create a new *Folder*.
3. Give the new folder the *Id* ‘ZopeZoo’.
4. Check *Create public interface*.
5. Click *Add*.

(For now, we will ignore the optional *Title* fields.)

### 7.11.3 Designing a Navigable Zoo

One of our goals is to enable easy user movement around the website. A key to this easy movement is a navigation interface that is consistent among the site’s pages. In other words, every web page in the site should present a similar set of hyperlinks, in a similar place on the page, on which users can rely to guide them through the site.

We also want to make sure the navigation links are always correct, regardless of how the structure of the site changes. The solution is to design a meaningful site structure, and then create the Zope methods that will dynamically present the current structure to web users in the form of navigation links.

First, let’s define the site structure. If the Zope Zoo was real, we might model the website’s organization on the zoo’s physical or logical design. For our purposes, we will pretend that the zoo houses three classes of animals. We’ll organize the website by adding folders inside our ZopeZoo folder.

### Step 2: Create Site Organization

1. Enter the *ZopeZoo* folder and create three subfolders with *Ids*: ‘Reptiles’, ‘Mammals’ and ‘Fish’.
2. Inside the *Mammals* folder, add one folder named ‘Whales’.
3. Navigate to the *Reptiles* folder and create two folders there: ‘Lizards’ and ‘Snakes’.

In Zope’s Navigator frame on the left side, you should see an icon for the ZopeZoo folder. (If you don’t see it, click *Refresh* in the Navigator.) To view the ZopeZoo folder hierarchy—i.e. our nascent web site’s structure—expand the ZopeZoo folder by clicking the little plus sign next to the icon. Similarly expand the zoo subfolders. You’ll see something like the figure below.

Now we create the basic presentation objects: The main template and the style sheet z_zoo.css. To get started, we ask a web designer to create a HTML mockup and a CSS file that together represent the web page layout shared throughout the site.

For the style sheet we create a simple *File* object in Zope. No need to make it dynamic.
Step 3: Create the Style Sheet

1. Go to the top level of our zoo website, the ZopeZoo folder.
2. Select File from the Add list.
3. Give the file an Id of 'z_zoo.css'.
4. Click Add.
5. Select z_zoo.css to get its Edit view.
6. Copy and paste these style definitions into the File Data area:

```html
body, p, th, td {
    font-family: Verdana, Arial, Helvetica, sans-serif;
    font-size: 10pt;
}

h1 {
    color: #6699cc;
    font-family: Verdana, Arial, Helvetica, sans-serif;
    font-size: 18pt;
    font-weight: bold;
}
p {
    color: #660000;
}

.status_message{
    background: #ffffaa;
    border-style: solid;
    border-width: thin;
    font-weight: bold;
}
```
At this stage, the HTML page the web designer created for us is valid XHTML 1.0 Strict and could also live in a static File object. But in the next steps we will convert the page into a dynamic template by adding TAL and METAL statements, so we need a Page Template object. For now we use the index_html method already added by selecting Create public interface in step 1.

**Step 4: Create the Main Template**

1. Select index_html to get its Edit view.
2. Replace all of the stock template code with this:

```html
<!DOCTYPE html PUBLIC
    "-//W3C//DTD XHTML 1.0 Strict//EN" "DTD/xhtml1-strict.dtd">
<html>
<head>
<title>PAGE TITLE OR ID</title>
<link rel="stylesheet" href="z_zoo.css" type="text/css" />
</head>
<body>
<div>&gt; <a href="ABSOLUTE_URL">PARENT TITLE OR ID</a> </div>
<ul>
<li><a href="ABSOLUTE_URL">SUB-OBJECT TITLE OR ID</a></li>
</ul>
<h1>PAGE TITLE OR ID</h1>
<p class="status_message">STATUS MESSAGE</p>
<p>THIS IS WHERE THE PAGE'S CONTENT GOES.</p>
</body>
</html>
```

Our web designer marked placeholders for dynamic elements with UPPERCASE letters. Using the Test tab of the new template, we can see the static HTML page. Don’t blame the web designer for the spartan layout. It’s for the sake of an easy example. If you don’t understand the XHTML and CSS code you might want to learn more about HTML first. This chapter shows you how to make that code dynamic.

**Step 5: Dynamic Title and Headline**

1. Go to the Edit tab of index_html.
2. Find these two lines:
The \textit{path expression} \texttt{context/title\_or\_id} returns the \textit{title} of the context object or - if that doesn’t exist - its \textit{id}. We work in the context of the \texttt{ZopeZoo} folder, which has no title. So clicking again on the \texttt{Test} tab you’ll see that title and headline are replaced by the id \texttt{ZopeZoo}. (You might want to open the \texttt{Test} tab in a new window to see the title of the browser window.) After completing the next step you’ll be able to navigate to subfolders and see title and headline change depending on the context.

\textbf{Step 6: Generate Subfolder Menu Dynamically}

1. Find the example menu item:

\begin{verbatim}
<ul>
  <li><a href="ABSOLUTE\_URL">SUB-OBJECT TITLE OR ID</a></li>
</ul>
\end{verbatim}

2. Extend it like this:

\begin{verbatim}
<ul tal:condition="python: context.objectValues(['Folder'])">
  <li tal:repeat="item python: context.objectValues(['Folder'])">
    <a href="ABSOLUTE\_URL"
        tal:attributes="href item/absolute\_url"
        tal:content="item/title\_or\_id">SUB-OBJECT TITLE OR ID</a>
  </li>
</ul>
\end{verbatim}

The \textit{Python expression} \texttt{context.objectValues(['Folder'])} returns all the subfolders in our context. The \texttt{tal:condition} statement checks if any subfolders exist. If not, the complete \texttt{ul} element is removed. That means we have reached a leaf of the navigation tree and don’t need a subfolder menu.

Otherwise, the same expression in the \texttt{tal:repeat} statement of the \texttt{li} element will return a list of subfolders. The \texttt{li} element will be repeated for each \texttt{item} of this list. In step 3 we created three subfolders in the \texttt{ZopeZoo} folder, so using again the \texttt{Test} tab we will see three list items, each with the correct id and link URL. For now there are no links back, so use the back button of your browser if you can’t wait exploring the site.

\textbf{Step 7: Generate Breadcrumbs Dynamically}

1. Look for this line:

\begin{verbatim}
<div>&gt; <a href="ABSOLUTE\_URL">PARENT TITLE OR ID</a></div>
\end{verbatim}

2. Replace it by:

\begin{verbatim}
<div tal:loop tal:repeat="item python: request.PARENTS[-2::-1]">&gt;
  <a href="ABSOLUTE\_URL"
      tal:attributes="href item/absolute\_url"
      tal:content="item/title\_or\_id">PARENT TITLE OR ID</a>
</div>
\end{verbatim}
Using a trail of bread crumbs for navigation is quite an old idea, you might remember Hansel and Gretel tried that to find their way home. In our days, breadcrumbs are used for site navigation and show the path back to the root (or home) of the site.

The folder that contains the current object is also called its parent. As long as we have not reached the root object, each folder has again a parent folder. ‘request.PARENTS’ is a list of all these parents from the current object down to the root object of the Zope application. ‘request.PARENTS[-2::-1]’ returns a copy of that list in reverse order, starting with the second last element. We don’t need the last value because ‘ZopeZoo’ is located in the second level of our Zope application and we just want to navigate within the zoo.

We use again a ‘tal:repeat’ statement to display the list. Because we don’t want to repeat the ‘div’ element, we add a dummy TAL element that doesn’t show up in the rendered HTML page. Now our site navigation is complete and you can explore the sections of the zoo.

### Step 8: Dynamic Status Bar

1. Go to this line:

   ```html
   <p class="status_message">STATUS MESSAGE</p>
   ```

2. Extend it by two tal attributes:

   ```html
   <p class="status_message" tal:condition="options/status_message | nothing"
       tal:content="options/status_message">STATUS MESSAGE</p>
   ```

We need the status bar later in this chapter. For now all we need is to make it invisible. ‘options/status_message’ will later be used for some messages. But most pages don’t have that variable at all and this path expression would raise an error. ‘options/status_message | nothing’ catches that error and falls back to the special value nothing. This is a common pattern to test if a value exists and is true.

### Step 9: Improve Style Sheet Link

1. Find this line in the HTML head:

   ```html
   <link rel="stylesheet" href="z_zoo.css" type="text/css" />
   ```

2. Replace it by:

   ```html
   <link rel="stylesheet" href="z_zoo.css" type="text/css"
       tal:attributes="href context/z_zoo.css/absolute_url" />
   ```

While the relative URI of the href attribute works thanks to acquisition, this isn’t a good solution. Using the index_html method for different folders, the browser can’t know that all the z_zoo.css files are in fact one and the same. Besides the CSS file the basic layout often contains a logo and other images, so making sure they are requested only once makes your site faster and you waste less bandwidth. The path expression ‘context/z_zoo.css/absolute_url’ returns the absolute url of the CSS file. Using it in the href attribute we have a unique URI independent of the current context.

### Step 10: Factor out Basic Look and Feel

1. Rename index_html to ‘z_zoo.pt’.

2. Wrap a ‘metal:define-macro’ statement around the whole page and add two ‘metal:define-slot’ statements for headline and content. After all these changes our main template - now called z_zoo.pt - looks like this:
3. Add again a new Page Template with the id ‘index_html’.

4. Replace the example code of index_html with these two lines:

```xml
<metal:macro metal:use-macro="context/z_zoo.pt/macros/page"/>
</metal:macro>
```

Transforming our main template into an external macro and including it again using the ‘metal:use-macro’ statement doesn’t change the resulting HTML page in any way. But in the next step we can add code we only want to use in index_html without changing the main template.

The ‘metal:define-macro’ statement in z_zoo.pt marks the complete template as reusable macro, giving it the id page. The expression ‘context/z_zoo.pt/macros/page’ in index_html points to that macro.
For later use we also added two ‘metal:define-slot’ statements within the macro. That allows to override headline and body while reusing the rest of the macro.

**Step 11: Add Special Front Page Code**

1. Go to the Edit tab of the new index.html.
2. Replace it by this code:

   ```
   <metal:macro metal:use-macro="context/z_zoo.pt/macros/page">
   <metal:slot metal:fill-slot="headline">
     <h1>Welcome to the Zope Zoo</h1>
   </metal:slot>
   <metal:slot metal:fill-slot="content">
     <p>Here you will find all kinds of cool animals. You are in the 
     <b tal:content="context/title_or_id">TITLE OR ID</b> section.</p>
   </metal:slot>
   </metal:macro>
   ```

The index.html should serve as the welcome screen for zoo visitors. In order to do so, we override the default slots. Take a look at how your site appears by clicking on the View tab of the ZopeZoo folder.

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.

### 7.11.4 Modifying a Subsection of the Site

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 main template including navigation.

**Step 12: Create index.html for the Reptile House**

1. Go to the Reptile folder.
2. Add a new Page Template named ‘index.html’.
3. Give it some content more appropriate to reptiles:

   ```
   <metal:macro metal:use-macro="context/z_zoo.pt/macros/page">
   <metal:slot metal:fill-slot="headline">
     <h1>The Reptile House</h1>
   </metal:slot>
   <metal:slot metal:fill-slot="content">
     <p>Welcome to the Reptile House.</p>
   </metal:slot>
   </metal:macro>
   ```

(continues on next page)
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 uses the same macro as the main index_html, 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.

### 7.11.5 Creating a File Library

File libraries are common on websites 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.

#### Step 13: Creating Library Folder and some Files

1. Add a new Folder to ZopeZoo with Id ‘Files’ and Title ‘File Library’.
2. Within that folder, add two File objects called ‘DogGrooming’ and ‘HomeScienceExperiments’.

We don’t need any content within the files to test the library. Feel free to add some more files and upload some content.

#### Step 14: Adding index_html Script and Template

1. Within the Files folder, add this new Script (Python) with the Id ‘index_html’:

```python
## Script (Python) "index_html"
##parameters=
##
library_items = []
items = context.objectValues(['File'])
for item in items:
    library_items.append({
        'title': item.title_or_id(),
        'url': item.absolute_url(),
        'modified': item.bobblebase_modification_time().aCommon()
    })

options = { 'library_items': tuple(library_items) }
return options
```

2. Also add a new Page Template named ‘index_html.pt’ with this content:

```metal
<table>
<tr><td>
</table>
</metal>"
This time the logic for our ‘index_html’ method will be more complex, so we should separate logic from presentation. We start with two unconnected objects: A Script (Python) to generate the results and a Page Template to present them as HTML page.

The script loops over `context.objectValues([‘File’])`, a list of all File objects in our Files folder, and appends for each file the needed values to the library_items list. Again the dynamic values are UPPERCASE in our mockup, so what we need are the file title, the url and the last modified date in a format like this: Mar 1, 1997 1:45 pm. Most Zope objects have the `bobobase_modification_time` method that returns a `DateTime` object. Looking at the API of `DateTime`, you’ll find that the `aCommon` method returns the format we want.

Later we will have more return values, so we store them in the `options` dictionary. Using the Test tab of the script you will see the returned dictionary contains all the dynamic content needed by our template.

The template uses again the page macro of `z_zoo.pt`. Unlike before there is only one ‘metal:fill-slot’ statement because we don’t want to override the headline slot. Go to the Test tab of the template to see how our file library will look like.

### Step 15: Bringing Things Together

1. Replace the last line of the `index_html` script by this one:

   ```python
   return getattr(context, ‘index_html.pt’)(**options)
   ```

2. Look for this example table row in `index_html.pt`:

   ```html
   <tr>
   <td><a href="URL">TITLE</a></td>
   <td>MON DD, YYYY H:MM AM</td>
   </tr>
   ```

3. Replace it by that code:

   ```html
   <tr tal:repeat="item options/library_items">
   <td><a href="URL" tal:attributes="href item/url" tal:content="item/title">TITLE</a></td>
   <td tal:content="item/modified">MON DD, YYYY H:MM AM</td>
   </tr>
   ```

Now our script calls the `index_html.pt` after doing all the computing and passes the resulting `options` dictionary to the template, which creates the HTML presentation of `options`. The Test tab of the template no longer works because it now depends on the script. Go to the Test tab of the script to see the result: The file library!

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.

---

**7.11. Creating Basic Zope Applications**

147
Step 16: Making the Library Sortable

1. Find the table headers in \textit{index\_html.pt}:

   \begin{verbatim}
   <th width=\"300\">File</th>
   <th>Last Modified</th>
   \end{verbatim}

2. Replace them with these dynamic table headers:

   \begin{verbatim}
   <th width=\"300\"><a href=\"SORT\_TITLE\_URL\"
   tal:omit-tag=\"not: options/sort_title_url\"
   tal:attributes=\"href options/sort_title_url\">
   File</a></th>
   <th><a href=\"SORT\_MODIFIED\_URL\"
   tal:omit-tag=\"not: options/sort_modified_url\"
   tal:attributes=\"href options/sort_modified_url\">
   Last Modified</a></th>
   \end{verbatim}

3. Extend \textit{index\_html} to make it look like this:

   \begin{verbatim}
   # Script (Python) \"index\_html\"
   ##parameters=sort='title'
   library_items = []
   items = context.objectValues(['File'])
   if sort == 'title':
      sort_on = ( ('title_or_id', 'cmp', 'asc'), )
      sort_title_url = ''
      sort_modified_url = '\$s?sort=modified' % context.absolute_url()
   else:
      sort_on = ( ('bobobase_modification_time', 'cmp', 'desc'), )
      sort_title_url = '\$s?sort=title' % context.absolute_url()
      sort_modified_url = ''
   items = sequence.sort(items, sort_on)
   for item in items:
      library_items.append(
         { 'title': item.title_or_id(),
           'url': item.absolute_url(),
           'modified': item.bobobase_modification_time().aCommon()
         })
   options = { 'sort_title_url': sort_title_url,
               'sort_modified_url': sort_modified_url,
               'library_items': tuple(library_items) }

   return getattr(context, 'index\_html.pt')(**options)
   \end{verbatim}

The changes in the template are quite simple. If an url is provided, the column header becomes a link. If not, the ‘not:’ expression of the ‘tal:omit-tag’ statement is true and the ‘a’ tag is omitted. The script will always provide an url for the column that isn’t currently sorted.

Basically we have to extend the logic, so most changes are in the script. First of all we define an optional parameter \textit{sort}. By default it is ‘title’, so if no value is passed in we sort by title. Sort criteria and urls depend on the sort parameter. We use the sort function of the built in \textit{sequence} module to apply the sort criteria to the \textit{items} list.

Now view the file library and click on the \textit{File} and \textit{Last Modified} links to sort the files. If there is a \textit{sort} variable and if it has a value of \textit{modified} then the files are sorted by modification time. Otherwise the files are sorted by \textit{title}. 

---

148 Chapter 7. The Zope Book
7.11.6 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, Page Templates, and other Zope services can be glued together to build simple applications.

7.11.7 Instance-Space Applications vs. Python packages

In contrast to building applications in instance space, you may also build applications in Zope by building them as Python packages. Building an application as a package differs from creating applications in instance space insasmuch as the act of creating a package typically is more familiar to developers and does not constrain them in any way.

Building a package 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”.

Building a package 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 package.

7.11.8 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 websites.

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.

7.12 Users and Security

Attention: This document was written for Zope 2.

7.12.1 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 website, 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.

### 7.12.2 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.

### 7.12.3 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.

### 7.12.4 Identification and Authentication

When a user accesses a protected resource (for example, by attempting to view a “protected” Page Template) Zope will ask the user to log in by presenting an authentication dialog. Once the dialog has been “filled out” and submitted, Zope will look for the user account represented by this set of credentials. By default Zope uses HTTP basic authentication. A cookie-based authentication can be implemented by adding a CookieCrumbler to the site’s base folder.
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 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.

### 7.12.5 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 “a folder object named zoo’ within the Zope root object”. 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 Page Template 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.

7.12.6 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.

Fig. 33: 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).

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 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. The most popular user folder implementation is called *PluggableAuthService* which allows you to mix-in and combine a vast number of different authentication schemes and backends, like LDAP or MySQL.
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 login 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 folders.

**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*. 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.

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.
7.12.7 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. The chapter of this book entitled Virtual Hosting provides some background that may be helpful to set up an SSL server in front of Zope.

7.12.8 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 a ‘myfolder’ folder that exists in the Zope root folder, that role cannot be used outside of the ‘myfolder’ 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 Page Templates in general if he does not possess a set of global roles which allow him to do so, but for Page Templates he owns, the user may edit the Page Template 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 permission 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 Page Templates and Folders, can be viewed. This action is protected by the View permission. Permissions are defined by Zope developers in Python packages and the Zope “core” itself. Packages 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. 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.

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 to the object’s Security tab. For example, click on the security tab of 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.

### 7.12.9 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.

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.

Let’s 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.zopezoo.org/Sales/manage.

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 websites 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.

7.12.10 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 website 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.

7.12.11 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 a Page Template:

```python
<form action="upload"
    tal:condition="python:
        modules['AccessControl'].getSecurityManager().checkPermission('Add / Documents, Images, and Files', context)">
    ...
</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:

```python
<form action="upload"
    tal:condition="python: context.check_security('Add Documents, Images and Files', here)">
    ...
</form>
```

Here is the ‘check_security’ Python Script which is referenced within the Page Template:

```python
## 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 Page Template:

```html
<p tal:content="user/.getUserName">username</p>
```

The Zope security API for Scripts is explained in Appendix B: API Reference. The Zope security API for Page Templates is explained in Appendix C: Zope Page Templates Reference.

### 7.12.12 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, 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 Script (Python) 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.

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

Python-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 a script, Zope runs the object’s executable content. The objects 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 Script (Python) 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 Script (Python) that sends email so that when it executes it has the “Manager” role. Visit the Proxy management tab on your Python script, as shown in the figure below.

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.

7.12.13 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.

7.13 Advanced Page Templates

**Attention:** This document was written for Zope 2.

In the chapter entitled Using Zope Page Templates you learned the basic features of Page Templates. In this chapter you’ll learn about advanced techniques including new types of expressions.

7.13.1 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 ‘&amp;lt;’. 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 by an object 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:

```xml
<p tal:repeat="newsItem context/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 ‘context’ refers to the folder in which the template is rendered; See the “Expressions” section further below in this chapter for more information on ‘context’. In this case, we use ‘context’ 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:

```xml
<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.

Default Content

You can leave the contents of an element alone by using the ‘default’ expression with ‘tal:content’ or ‘tal:replace’. For example:

```xml
<p tal:content="default">Spam</p>
```

This renders to:
Most often you will want to selectively include default content, rather than always including it. For example:

```
<p tal:content="python:context.getFood() or default">Spam</p>
```

**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.

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 context/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:
This example uses Python expressions, which are covered later in this chapter.

One 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:

```
<table tal:define="objects context/objectValues;"
   sort_on python:('title', 'nocase', 'asc'),;
   sorted_objects python:sequence.sort(objects, sort_on)">
   <tr tal:repeat="item sorted_objects">
       <td tal:content="item/title">title</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. For example, the code below will generate an “href” and a “class” attribute:

```
<a href="link"
   tal:attributes="href context/getLink;
   class context/getClass">link</a>
```

You can also define attributes with XML namespaces. For example:

```
<Description
   dc:Creator="creator name"
   tal:attributes="dc:Creator context/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:

```
<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.

One 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:

```
<table tal:define="objects context/objectValues;"
   sort_on python:('title', 'nocase', 'asc'),;
   sorted_objects python:sequence.sort(objects, sort_on)">
   <tr tal:repeat="item sorted_objects">
       <td tal:content="item/title">title</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. For example, the code below will generate an “href” and a “class” attribute:

```
<a href="link"
   tal:attributes="href context/getLink;
   class context/getClass">link</a>
```

You can also define attributes with XML namespaces. For example:

```
<Description
   dc:Creator="creator name"
   tal:attributes="dc:Creator context/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:
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:

```xml
<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. To have it available on both tags, you can place the definition on some element that encloses both the ‘h4’ and the ‘ul’ for example the ‘body’.

You can define more than one variable using `tal:define` by separating them with semicolons. For example:

```xml
<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:

```xml
<p tal:define="title template/title; untitled not:title; tlen python:len(title);" />
```

With judicious use of ‘tal:define’ you can improve the efficiency and readability of your templates.

### Omitting Tags

You can remove tags with the `tal:omit-tag` statement. You will seldom need to use this TAL statement, but occasionally it’s useful. The omit-tag attribute removes opening and closing tags, but does not affect the contents of the element. For example:

```xml
<b tal:omit-tag=""><i>this</i> stays</b>
```

Renders to:

```xml
<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="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="prefs context/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 context/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:

```python
## 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>"
```

(continues on next page)
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:

```xml
<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 context/getItems">
    <tal:items repeat="item items">
        <tal:parts repeat="part item">
            <p tal:content="part">Part</p>
        </tal:parts>
    </tal: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**

With Zope Page Templates you can use the form/action/response pattern. 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.

For example here’s a part of a form template:
This form could be processed by this script:

```python
## 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:

```python
## 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 ‘context’ 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:

```html
<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. You can use one of any number of hacks to call an input processing method without inserting its results. For example:

```html
<span tal:define="unused context/processInputs"
    tal:omit-tag=""/>
```
This sample calls the ‘processInputs’ method and assigns the result to the ‘unused’ variable.

### 7.13.2 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.

- **‘context’** 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.

- **‘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 ‘context’ 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 ‘context’ 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:
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 ‘&quot;’. Since dollar signs are used to signal path expressions, a literal dollar sign must be written as two dollar signs (‘$$’). For example:

"string:Please pay $$dollars_owed"
"string:She said, &quot;Hello world.&quot;"

Some complex string formatting operations (such as search and replace or changing capitalization) can’t easily be done with string expressions. For these cases, you should use Python expressions or Scripts.

**Path Expressions**

Path expressions refer to objects with a path that resembles a URL path. A path describes a traversal from object to object. All paths begin with a known object (such as a built-in variable, a repeat variable, or a user defined variable) and depart from there to the desired object. Here are some example paths expressions:

```python
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 ‘context/x’ instead. You can do this by listing the paths in order of preference, separated by vertical bar characters (‘|’):

```html
<h4 tal:content="request/form/x | context/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:

```html
<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:

```html
<p tal:condition="not:context/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 page template into a variable so that you can refer to its properties, you can’t use a normal path expression because it will render the template 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:

```html
<span tal:define="page nocall:context/aPage">
  <p tal:content="string:${page/getId}: ${page/title}"
    tal:define="id nocall:context/aPage/getId">
    Id: Title
  </p>
</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:

```html
<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:

```html
<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:

```html
<h4 tal:condition="exists:request/form/errmsg"
   tal:content="request/form/errmsg">Error!</h4>
```

You can combine exists expressions with not expressions, for example:

```html
<p tal:condition="not:exists:request/form/errmsg" 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:

```html
<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 and or operators, like this:

```html
You <span tal:define="name user/getUserName"
   tal:replace="python:name=='Anonymous User' and
   'need to log in' or default">are logged in as
```

(continues on next page)
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 . . . ’.

This operator combination works like an if/then/else statement. Here’s another example of how you can use this pattern:

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 this pattern you could also 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:

The final example has a slightly different meaning than the path expression, “request/form/x | default”, since it will use the default text if “request/form/x” doesn’t exists or if it is false.

Getting at Zope Objects

Much of the power of Zope involves tying together specialized objects. Your Page Templates can use Scripts, SQL Methods, Catalogs, and custom content objects. In order to use these objects you have to know how to get access to them within Page Templates.

Object properties are usually attributes, so you can get a template’s title with the expression “template.title”. Most Zope objects support acquisition, which allows you to get attributes from “parent” objects. This means that the Python expression “context.Control_Panel” will acquire the Control Panel object from the root Folder. Object methods are attributes, as in “context.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:

Instead, you must write:

since Python doesn’t support attribute names with periods.

Some objects, such as ‘request’, ‘modules’, and Zope Folders support Python item access, for example:
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:

"context/images/penguin.gif"

instead of:

"python:getattr(context.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:context.myscript(1, 2)"
"python:context.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:
Name is 30 years old.

Of course, it’s fine to return Zope objects and Python lists as well.

**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 variable for it, like this:

```python
tal:define="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’ namespace package. In particular, the “standard” module in this package provides a number of useful formatting functions. The full name of this module is “Products.PythonScripts.standard”, so you could get access to it using either of the following statements:

```python
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 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”.

### 7.13.3 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 Zope Services.

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:
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:

```python
for i in range(250):
    for j in range(250):
        for k in range(250):
            pass
return 'Done'
```

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.

### Filesystem caching for Chameleon-based templates

Zope 4 introduced the Chameleon HTML/XML template engine as new backend for Zope Page Templates. The Chameleon templating engine can compile templates and cache them on the file system for faster startup and execution.

File system caching is activated by setting an environment variable named CHAMELEON_CACHE to the path of a folder on the filesystem where Chameleon can write its compiled template representation.

Look for or add a section named environment in etc/zope.conf and add a suitable filesystem path, for example:

```xml
<environment>
    CHAMELEON_CACHE $INSTANCE/var/cache
</environment>
```

Make sure that folder exists before starting Zope.

How to configure Zope is explained in Configuring Zope.
7.13.4 Page Template Utilities

Zope Page Templates are powerful but simple. They 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. 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.

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:

```python
<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:

```python
<ul tal:define="lots python:range(100); batch python:modules['ZTUtils'].Batch(lots,
 size=10,
 start=13)">
    <li tal:repeat="num batch"
        tal:content="num">0</li>
</ul>
```

This batch starts with the fourteenth item and ends with the twenty third item. In other words, it displays the numbers 13 through 22. It’s important to notice that the batch ‘start’ argument is the index of the first item. Indexes count from zero, rather than from one. So index 13 points to the fourteenth item in the sequence. Python uses indexes to refer to list items.

Normally when you use batches you’ll want to include navigation elements on the page to allow users to go from batch to batch. Here’s a full-blow batching example that shows how to navigate between batches:

```html
<html>
<head>
    <title tal:content="template/title">The title</title>
</head>
<body tal:define="employees context/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>
(continues on next page)
```
Define a Script (Python) with the name getEmployees in the same folder with the following body (no parameters are necessary):

```python
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 or the current page (`request/URL0`) along with a query string indicating the start index of the batch. For example, if the current batch starts with index 10, then the previous batch will start with an index of 0. The `first` variable of a batch gives its 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 in 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.

7.13.5 Conclusion

This chapter covers some useful and some obscure nooks and crannies of Page Templates, and after reading it you may feel a bit 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.

7.14 Advanced Zope Scripting

Attention: This document was written for Zope 2.

In the chapter entitled “Basic Zope Scripting”, you have seen how to manage Zope objects programmatically. In this chapter, we will explore this topic some more. Subjects discussed include additional scripting objects, script security, and calling script objects from presentation objects like Page Templates. As we have mentioned before, separation of logic and presentation is a key factor in implementing maintainable web applications.

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 Page Templates to handle presentation, and Zope scripting to handle logic.

7.14.1 Warning

Zope Script objects are objects that encapsulate a small chunk of code written in a programming language. They first appeared in Zope 2.3, and have been the preferred way to write programming logic in Zope for many years. Today it is discouraged to use Scripts for any but the most minimal logic. If you want to create more than trivial logic, you should approach this by creating a Python package and write your logic on the file system.

This book does not cover this development approach in its details. This chapter is still useful to read, as it allows you to get an understanding on some of the more advanced techniques and features of Zope.

7.14.2 Calling Scripts

In the “Basic Zope Scripting” chapter, you learned how to call scripts from the web and, conversely, how to call Page Templates from Python-based Scripts. In fact scripts can call scripts which call other scripts, and so on.
Calling Scripts from Other Objects

You can call scripts from other objects, whether they are Page Templates or Scripts (Python). 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 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:

```html
<div tal:replace="context/hippo/feed">
  Output of feed()
</div>
```

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.

To call a script without inserting a value in the page, you can use `define` and ignore the variable assigned:

```html
<div tal:define="dummy context/hippo/feed" />
```

In a page template, `context` 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:

```html
<div tal:replace="python:context.hippo.feed(food='spam')">
  Output of feed(food='spam')
</div>
```

Just as in Path Expressions, the ‘context’ 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).

One difference is the notation used for attribute access – Script (Python) uses the standard Python period notation, whereas in a TALES path expression, a forward slash is used.

For further reading on using Scripts in Page Templates, refer to the chapter entitled Advanced Page Templates.

Calling scripts from Python

Calling scripts from other scripts works similar to calling scripts from page templates, except that you must always use explicit calling (by using parentheses). For example, here is how you might call the `updateInfo` script from Python:

```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.

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. The figure below shows a slightly updated object hierarchy that contains a script named `vaccinateHippo.py`.

Here is how you can call the `vaccinate` script on the `hippo` object from the `vaccinateHippo.py` script:
Fig. 40: A collection of objects and scripts
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.

**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 and from Page Templates.

- by URL:
  
  ```
  ```

- from a Script (Python):
  
  ```
  context.foo.updateInfo(amount="lots")
  ```

- from a Page Template:
  
  ```
  <span tal:content="context/foo/updateInfo" />
  ```

- from a Page Template, with arguments:
  
  ```
  <span tal:content="python:context.foo.updateInfo(amount='lots')" />
  ```

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:

```python
updateInfo = getattr(context, "updateInfo", None)
if updateInfo is not None:
    updateInfo(color="brown", pattern="spotted")
else:
    # complain about missing script
    return "error: updateInfo() not found"
```

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.

### 7.14.3 Using External Methods

Sometimes the security constraints imposed by Python-based 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”:InstallingZope.html 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:

```python
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.

![Fig. 41: External Method Properties view](image)

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:

```python
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):

```python
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:

```python
## 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.

**Creating Thumbnails from Images**

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:

```python
def makeThumbnail(self, original_id, size=200):
    """
    (continues on next page)
```
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.

```python
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”:http://www.pythonware.com/products/pil/index.htm for more information on...
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 Image-Folder/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:

```python
## Script (Python) "makeThumbnails"
##
for image_id in context.objectIds('Image'):
    context.makeThumbnail(image_id)
```

This will loop through all the images in a folder and create a thumbnail for each one.

Now call this script on a folder with images in it. It will create a thumbnail image for each contained image. Try calling the makeThumbnails script on the folder again and you’ll notice it created thumbnails of your thumbnails. This is no good. You need to change the makeThumbnails script to recognize existing thumbnail images and not make thumbnails of them. Since all thumbnail images have an original_id property you can check for that property as a way of distinguishing between thumbnails and normal images:

```python
## 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:

```dtm1
<dtml-var standard_html_header>

<dtml-if updateThumbnails>
    <dtml-call makeThumbnails>
</dtml-if>

<h2>Thumbnails</h2>
<table><tr valign="top">
    <dtml-if expr="objectValues('Image')">
        <dtml-if original_id>
            <td>
                <a href="&dtml-original_id;">dtml-var sequence-item</a><br/>
            </td>
        </dtml-if>
    </dtml-if>
</tr></table>
<form>
```

(continues on next page)
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.

**Fig. 42: 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”:http://www.jabber.com/ 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
# 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)
```

(continues on next page)
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. 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”:http://www.zope.org/Documentation/Books/ZDG/current.

### 7.14.4 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 hierarchy 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.

The most important thing for you to understand in this chapter is that context acquisition exists and can interfere with whatever you are doing. Today it is seen as a fragile and complex topic and rarely ever used in practice.

Recall our Zoo example introduced earlier in this chapter.

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 kangarooMouse object you should use the URL Zoo/Vet/SmallAnimals/kangarooMouse/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:

![Fig. 44: Acquisition example folder structure](image)

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 hierarchy 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 hierarchy 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 DTM 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.

### 7.14.5 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:

```html
<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:

```python
## 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: API Reference.

In the Script (Python) 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. You could manually convert the string to an integer using the Python `int` built-in:

```python
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:

```html
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.

*Fig. 45: Parameter conversion error*

Zope Error

Zope has encountered an error while publishing this resource.

**Error Type:** ValueError

**Error Value:** An integer was expected in the value '22 going on 23'

**Troubleshooting Suggestions**

- The URL may be incorrect.
- The parameters passed to this resource may be incorrect.
- A resource that this resource relies on may be encountering an error.

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

If the error persists please contact the site maintainer. Thank you for your patience.

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.

text  Converts a string to a list by breaking it on new lines.

date  Converts a string to a \texttt{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:

\begin{verbatim}
<form action="processTimes">
  <p>I would prefer \textbf{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>
\end{verbatim}

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" />
  Last Name  <input type="text" name="person.lname:record" />
  Age  <input type="text" name="person.age:record:int" />
  <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:

```python
## 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, \texttt{int:list} is identical to \texttt{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:

\textbf{\texttt{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 \texttt{action} is \texttt{method}.

\textbf{\texttt{default_action}} Appends the attribute value to the original action of the form when no other \texttt{action} converter is used.

Here’s an example form that uses action converters:

```xml
<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>
```

We assume a folder ‘employeeHandlers’ containing two scripts named ‘fireEmployees’ and ‘promoteEmployees’. The form will call either the \texttt{fireEmployees} or the \texttt{promoteEmployees} script, depending on which of the two submit buttons is used. Notice also how it builds a list of employees with the \texttt{list} converter. Form converters can be very useful when designing Zope applications.

### 7.14.6 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.

The chapter entitled \textit{Users and Security} covers security in more detail. You should consult the \textit{Roles of Executable Objects} and \textit{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 unauthorized changes to Zope objects and accessing the server Zope is running on. These restrictions are implemented through a collection of limits on what your scripts can do. The limits are not effective enough to prevent malicious users from harming the Zope process on purpose. They only provide a safety belt against accidental bad code.

**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 \texttt{for} and \texttt{while} loops. The reason for this restriction is to limit your ability to hang Zope by creating an infinite loop.

This limit does not protect you from creating other sorts of infinite recursions and it’s still possible to hang the Zope process.

**Import limits** Scripts cannot import arbitrary packages and modules. You are limited to importing the \texttt{Products.PythonScripts.standard} utility module, the \texttt{AccessControl} module, some helper modules (\texttt{string}, \texttt{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.

**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 packages 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. You probably should not grant access to scripts to untrusted people.

### 7.14.7 Python versus Page Templates

Zope gives you multiple ways to script. For small scripting tasks the choice of Python-based Scripts or Page Templates probably doesn’t make a big difference. For larger, logic-oriented tasks you should use Python-based Scripts or write packages on the file-system.

For presentation, Python should *not* be used; instead you use ZPT.

Just for the sake of comparison, here is a simple presentational script suggested by Gisle Aas in ZPT and Python.

In ZPT:

```xml
<%var item, context%>
<div tal:repeat="item context/objectValues"
    tal:replace="python:'%s
    %s
' % (item.getId(), str(item))" />
```

In Python:

```python
for item in context.objectValues():
    print "%s: %s" % (item.getId(), item)
print "done"
return printed
```

### 7.14.8 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 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, Java and JavaScript.

In-depth information on XML-RPC can be found at the “XML-RPC website”:http://www.xmlrpc.com/.

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:

```python
import xmlrpclib

server = xmlrpclib.Server('http://www.zopezoo.org/)
for employee in server.JanitorialDepartment.personnel():
    server.fireEmployee(employee)
```

In Java:

```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.

**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 would 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 username and password with `wget` to access protected scripts:

```
```

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
```

(Beware of the linebreak – the above should be input as one line, minus the backslash).

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 [8-9].

Word will then load the URL and return the results of calling the Zope script. Despite the fact that Word doesn’t let you POST arguments this way, you can pass GET arguments by entering them as part of the URL.

You can even control this behavior using Word’s built-in Visual Basic scripting. For example, here’s a fragment of Visual Basic that tells Word to open a new document using a Zope script URL:

```
Documents.Open FileName:="http://www.zopezoo.org/LionCages/wash?use_soap=1&water_temp=hot"
```

You could use Visual Basic to call Zope script URLs in many different ways.

Zope’s URL to script call translation is the key to remote scripting. Since you can control Zope so easily with simple URLs you can easy script Zope with almost any network-aware system.

### 7.14.9 Conclusion

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.
7.15 Zope Services

**Attention:** This document was written for Zope 2.

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.

### 7.15.1 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:

```python
useragent = request.get('HTTP_USER_AGENT', '')
if useragent.find('Windows') != -1:
    request.set('OS', 'Windows')
elif useragent.find('Linux') != -1:
    request.set('OS', 'Linux')
```

(continues on next page)
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 page template named ‘test’ in the accessrule_test folder with the following text:

```html
<html>
<body>
<pre tal:content="context/REQUEST">request details</pre>
</body>
</html>
```

Save the ‘test’ page template 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’.

### 7.15.2 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.

### 7.15.3 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 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 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. Scripts can get this complex, especially if you use lots of looping (such as 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 website 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 website 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 Scripts. 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 page, create a new page template object in the root folder called Weather. This object will contain some weather information. For example, let’s say it contains:

```html
<html>
<body>
  <p>Yesterday it rained.</p>
</body>
</html>
```

Now, click on the Weather page template and click on its Cache view. This view lets you associate this page 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 page, they will get the cached copy instead. For a page 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:

```html
<html>
<body>
  <p>Yesterday's weather was
  <tal:yesterday tal:replace="context/yesterdayQuery" />
</p>

  <p>The current temperature is
  <tal:current tal:replace="context/currentTempQuery" />
</p>
</body>
</html>
```

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 7.15. Zope Services
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 page should be cached, however, then the page will only make the query when the cache expires. The default cache time is 300 seconds (5 minutes), so setting this page 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.

7.15.4 Outbound Mail Services

Zope comes with an object that is used to send outbound e-mail, usually in conjunction with a Script (Python).

Mailhosts can be used Python 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 Script.

7.15.5 Error Logging Services

Note: As of Zope 4, the Site Error Log is no longer a dependency of Zope, but has to be installed separately if you want to use it.

The Site Error Log is available as Products.SiteErrorLog on PyPI.

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.

Available options for the Site Error Log instance:

- Number of exceptions to keep - This option is set to 20 by default, rotating old exceptions out when more than 20 are stored. You may set this to a higher or lower number as you like.

- Copy exceptions to the event log - If this option is enabled, the Site Error Log object will copy the text of the received exceptions to Zope’s event log.

- Ignored exception types - Here you can add Exceptions which you want to ignore. This means they will be neither shown in the ZMI, nor logged to Zope’s event log. By default Unauthorized, NotFound and Redirect are set.

7.15.6 Virtual Hosting Services

For detailed information about using virtual hosting services in Zope, see the chapter entitled Virtual Hosting Services.
7.15.7 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.

7.15.8 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.

7.16 Basic DTML

**Attention:** This document was written for Zope 2.

**Note:** DTML has been the primary markup language within Zope for a long time. However the recommended primary markup language within Zope is nowadays ZPT (Zope Page Templates). ZPT is your choice for generating markupish output like HTML or XML. The usage of DTML should be limited where you have to generate non-markupish output like text files or other formats. Since DTML is pretty old it really does not support features like internationalization or unicode very well. In addition the syntax of DTML is not always very easy to understand. You have to learn DTML to some point if you intend to use ZSQL methods (for RDBMS integration with Zope) - but even for the RDBMS integration we have better solutions like Object-Relational-Mappers (check with the chapter about relational database connectivity).

DTML (Document Template Markup Language) is a templating facility which supports the creation of dynamic HTML and text. In Zope it is most often used when you want to generate non-HTML or non-XML content, like parts of SQL queries, dynamic CSS and JavaScript files or email templates. Generating HTML and XML is usually done with page templates inside Zope.

DTML is a *tag-based* presentation and scripting language. This means that *tags* (e.g. `<dtml-var name=''>`) embedded in your text cause parts of your text 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.

### 7.16.1 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 a page. DTML does allow you to embed Python *expressions* (a == 1) into tags. It provides flow control and conditional logic by way of “special” 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.

### 7.16.2 When To Use DTML

If you don’t want to use page templates for whatever reason DTML might work well. Likewise, if you want to dynamically create non-HTML text (like CSS stylesheets or email messages), DTML can help.
7.16.3 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 output some text 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 output 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 some dynamic output 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, by using functions that manipulate strings, you are better off doing it in Python, which has more powerful string processing capabilities than DTML.

7.16.4 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” text document, you might create a DTML Document object to hold the 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!

7.16.5 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:
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.

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

### 7.16.6 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 a text inside the dtml-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.

### 7.16.7 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:

```html
<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.

### 7.16.8 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.

### 7.16.9 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 content 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”. Note: an ‘id’ is how you refer to an object such as a DTML Method or a Folder later on; titles are for informational purposes only.

Change the contents of the DTML Method to the following:

```html
<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:

If you tell your browser to view the HTML source of the Workspace frame, you will see something not unlike the below:
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 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 Variables and 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 named “actors” in your sandbox folder. Give the script the following body and save it:

```python
## 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. Don’t worry about the “##’s for now, we
will explain these later.

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
<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
<html>
<body>
<h1>Actors in the movie Glengarry Glen Ross</h1>
<table border="1">
  <thead>
    <tr>
      <th>Name</th>
    </tr>
  </thead>
  <tbody>
    <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>
  </tbody>
</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:

```html
<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:

```xml
<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:

![Fig. 48: Result of submitting the infoForm method](image-url)

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 following body:

```xml
<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:

```xml
<dtml-var standard_html_header>
<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/VYHU0ALYElA'

lazy items
SESSION <bound method SessionDataManager.getSessionData of SessionDataManager>,
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/VYHU0ALYElA'
URL 'http://localsaints:8084/DTML_Exmaple/show_request'
AUTHENTICATED_USER admin
TraversalRequestNameStack [],
URL0 http://localsaints:8084/DTML_Exmaple/show_request
URL1 http://localsaints:8084/DTML_Exmaple
URL2 http://localsaints:8084
BASE0 http://localsaints:8084
BASE1 http://localsaints:8084
BASE2 http://localsaints:8084/DTML_Exmaple
BASE3 http://localsaints:8084/DTML_Exmaple/show_request

environ
SCRIPT_NAME '',
HTTP_ACCEPT_ENCODING 'gzip, deflate, compress;q=0.9'
SERVER_PORT '8084'
```

(continues on next page)
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.

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:
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:

```
<dtm-var standard_html_header>
<p>Welcome</p>
<dtm-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.

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>
```

(continues on next page)
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:

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 website. 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
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 website.

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:

```python
## 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:

```dtm VAR
<dtm-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:

```dtm VAR
<dtm-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:

```dtm IF
<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:

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

and:

```html
<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:

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

will call the method. However:

```html
<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:

```html
<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:

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

with:

```html
<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.

7.16.10 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 all string values which are retrieved from the REQUEST namespace are HTML-quoted by default. This helps to prevent “cross-site scripting” security holes, 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:

  ```html
  <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:

  ```html
  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:

```html
&dtml-cockatiel;
```

This is equivalent to:

```html
<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:

```html
<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.
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:

```html
<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 can be nested to any depth, for example, you could have:

```html
<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 they evaluate the second condition. 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:

```html
<dtml-if monkey_house>
    <p>There *is* 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:

```html
<dtml-if expr="monkey_house">
    <p>There *is* 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:

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

```html
<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:

```xml
<dtml-if hasVisitedZoo>
  <p>Zoo admission <dtml-var adult_rate fmt="dollars-and-cents">.</p>
</dtml-if>

<dtml-else>
  <p>Zoo admission for first time visitors
  <dtml-var expr="adult_rate/2" fmt="dollars-and-cents"></p>
</dtml-else>
```

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:

```python
## 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:

```xml
<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 \texttt{in} tag block is a kind of \textit{template} that is applied once for each item in a sequence.

\section*{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:

\begin{verbatim}
<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>
\end{verbatim}

This code displayed the following file listing, as shown in the figure below.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{zope-listing.png}
\caption{Iterating over a list of files}
\end{figure}

Let's look at this DTML example step by step. First, the \texttt{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 \texttt{ul} HTML tag.

Then there is the \texttt{in} tag. The tag has an expression that is calling the Zope API method called \texttt{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 \texttt{in} tag will loop over every item in this sequence. If there are four file objects in the current folder, then the \texttt{in} tag will execute the code in its block four times; once for each object in the sequence.
During each iteration, the \textit{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 \textit{in} tag iterates over a collection of File objects and uses the \textit{var} tag to look up variables in each file:

\begin{verbatim}
<dtml-in expr="objectValues('File')"
   <li><a href="&dtml-absolute_url;"><dtml-var title_or_id></a></li>
</dtml-in>
\end{verbatim}

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

\begin{verbatim}
<ul>
  <li><a href="http://localhost:8080/FirstFile">FirstFile</a></li>
</ul>
\end{verbatim}

During the second iteration the second object’s \textit{absolute_url} and \textit{title_or_id} variables are inserted in the output:

\begin{verbatim}
<ul>
  <li><a href="http://localhost:8080/FirstFile">FirstFile</a></li>
</ul>
\end{verbatim}

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

\textbf{In Tag Special Variables}

The \textit{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.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure53}
\caption{File listing with alternating row colors}
\end{figure}

7.16. Basic DTML
The \textit{in} tag makes this easy. Change your file library method a bit to look like this:

\begin{verbatim}
<dtml-var standard_html_header>
<table>
<dtml-if expr="objectValues('File')">
  <dtml-if sequence-even>
    <trbgcolor="grey">
  </dtml-if>
  <tr>
    <td>
      <a href="&dtml-absolute_url;"><dtml-var title_or_id></a>
    </td>
  </tr>
</dtml-in>
</table>
<dtml-var standard_html_footer>
\end{verbatim}

Here an \textit{if} tag is used to test for a special variable called `sequence-even`. The \textit{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 \textit{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 \textit{in} tag defines for you. Here are the most common and useful:

\textbf{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.

\textbf{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.

\textbf{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.

\textbf{sequence-start}  This variable is true for the very first iteration.

\textbf{sequence-end}  This variable is true for the very last iteration.

These special variables are detailed more thoroughly in Appendix A.

\subsection*{7.16.11 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.
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:

```html
<dtml-var standard_html_header>

<dtml-if zooName>
  <p><dtml-var zooName></p>
</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.

7.17.1 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 popping objects on to and off of the stack.

When a DTML Document or DTML Method is executed, Zope creates a DTML namespace to resolve DTML variable names. It’s important to understand the workings of the DTML namespace so that you can accurately predict how Zope will locate variables. Some of the trickiest problems you will run into with DTML can be resolved by understanding the DTML namespace.

When Zope looks for names in the DTML namespace stack it first looks at the 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.

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.

7.17.2 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.

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:
Zope Error

Zope has encountered an error while publishing this resource.

Error Type: KeyError
Error Value: unicorn

Troubleshooting Suggestions

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

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

Fig. 54: DTML error message indicating that it cannot find a variable

Fig. 55: Initial DTML namespace stack
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:

```dtmldoc
<ul>
  <li><dtml-var title_or_id></li>
</ul>
```

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. 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. Variables explicitly set on the request.
2. Special variables. The REQUEST namespace provides you with special information, such as the URL of the current object and all of its parents.
3. 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.
4. 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.

5. Cookies. If the client of the current request has any cookies these can be found in the current REQUEST object.

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:

```xml
<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.

![Fig. 57: Displaying the request](image)

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.

### 7.17.3 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.

7.17.4 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:

```html
<dtm-var standard_html_header>
<dtm-var expr="Reptiles.getReptileInfo()">
<dtm-var expr="Reptiles.reptileHouseMaintainer">
<dtm-in expr="Reptiles.getReptiles()">
  <dtm-var species>
</dtm-in>
</dtm-in>
<dtm-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:

```html
<dtm-var standard_html_header>
<dtm-with Reptiles>
  <dtm-var getReptileInfo>
    <dtm-var reptileHouseMaintainer>
      <dtm-in getReptiles>
        <dtm-var species>
      </dtm-in>
    </dtm-var>
  </dtm-var>
</dtm-with>
<dtm-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:

```html
<dtm-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:

```html
<dtm-with expr="REQUEST.form">
  <dtm-var id>
</dtm-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:

```html
<dtm-with expr="REQUEST.form" only>
  <dtm-if id>
    <dtm-var id>
  </dtm-if>
  <dtm-else>
    <p>The form didn't contain an "id" variable.</p>
</dtm-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:

```html
<dtml-let person="'Bob'", relation="'uncle'">
  <p><dtml-var person>'s your <dtml-var relation>.</p>
</dtml-let>
```

This would display:

```html
<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.

#### 7.17.5 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:

```html
<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?:

```python
<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:

```python
<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:

```python
<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 getattr utility function to explicitly look up a variable:
The square of <dtml-var sequence-item> is:

\[ \text{\_getitem('sequence-item')} \times \text{\_getitem('sequence-item')} \]

The \text{\_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 \text{\_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 \text{\_getitem} function does not render a variable.

Here’s how to insert a rendered variable named \text{myDoc}:

\[ \text{\_getitem('myDoc', 1)} \]

This example is in some ways rather pointless, since it’s the functional equivalent to:

\[ \text{\_getitem('myDoc')} \]

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 \text{selectedDoc} which contained the name of the document. You could then display the rendered document like so:

\[ \text{\_getitem(selectedDoc, 1)} \]

Notice in the above example that \text{selectedDoc} is not in quotes. We don’t want to insert the text \text{selectedDoc} we want to insert the value of the variable named \text{selectedDoc}. For example, the value of \text{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 \text{from DTML}. This is explained more in the chapter entitled \text{Advanced Zope Scripting}. Using Python sidesteps many of the issues in DTML.

### 7.17.6 DTML Security

Zope can be used by many different kinds of users. For example, the Zope site, \text{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 \text{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:

\[ \text{\_getitem('Diary')} \]

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 \text{Unauthorized} error and the user will be asked to present more privileged authentication credentials.

In the chapter entitled \text{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.
7.17.7 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.

7.17.8 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 website, 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.

7.17.9 The \texttt{Call} Tag

The \texttt{var} tag can call methods, but it also inserts the return value. Using the \texttt{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, \texttt{animalName}, you are more interested in the effect of calling the \texttt{manage\_changeProperties} method than the return value the method gives you. Here’s an example:

```html
<dtml-if expr="REQUEST.has_key('animalName')">
  <dtml-call expr="manage\_changeProperties(\texttt{animalName}=REQUEST['animalName'])">
    <h1>The property \texttt{animalName} has changed</h1>
  </dtml-call>
</dtml-if>
<dtml-else>
  <h1>No properties were changed</h1>
</dtml-else>
```

In this example, the page will change a property depending on whether a certain name exists. The result of the \texttt{manage\_changeProperties} method is not important and does not need to be shown to the user.

Another common usage of the \texttt{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 \texttt{let} tag:

```html
<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 \texttt<a href="<dtml-var target>"><dtml-var target></a>.</h1>
</dtml-let>
<dtml-call expr="RESPONSE.redirect(target)"
```

(continues on next page)
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:

```html
<dtml-call "insertLogEntry(REQUEST)">
```

or:

```html
<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:

```html
<dtml-call "preprocess(REQUEST)">
```

### 7.17.10 The Comment Tag

DTML can be documented with comments using the `comment` tag:

```html
<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>
<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.
7.17.11 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.

![Fig. 58: HTML tree generated by the tree tag](image)

Here’s the DTML that generated this tree display:

```dtmlext
<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:

```python
## 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:

```xml
<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.

For a summary of the `tree` tag arguments and special variables see Appendix A.
7.17.12 The \textit{Return} Tag

In general DTML creates textual output. You can however, make DTML return other values besides text. Using the \texttt{return} tag you can make a DTML Method return an arbitrary value just like a Python or Perl-based Script.

Here’s an example:

\begin{verbatim}
<p>This text \textit{is} ignored.</p>
<dtml-return expr="42">
\end{verbatim}

This DTML Method returns the number 42.

Another upshot of using the \texttt{return} tag is that DTML execution will stop after the \texttt{return} tag.

If you find yourself using the \texttt{return} tag, you almost certainly should be using a Script instead. The \texttt{return} tag was developed before Scripts, and is largely useless now that you can easily write scripts in Python and Perl.

7.17.13 The \textit{Sendmail} Tag

The \textit{sendmail} tag formats and sends a mail messages. You can use the \textit{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 \textit{sendmail} tag:

\begin{verbatim}
<dtml-sendmail>
To: <dtml-var recipient>
From: <dtml-var sender>
Subject: Make Money Fast!!!!
\end{verbatim}

(continues on next page)
Take advantage of our exciting offer now! Using our exclusive method you can build unimaginable wealth very quickly. Act now!

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.

### 7.17.14 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:

```html
<dtm-var standard_html_header>
<p>Send your 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:
<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>
<dtm-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:

```html
<dtm-var standard_html_header>
<dtm-if send_to>
<dtm-in send_to>
<dtm-sendmail smtphost="my.mailserver.com">
To: <dtm-var sequence-item>
Subject: Resume
</dtm-sendmail>
```

(continues on next page)
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.

### 7.17.15 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 DTMIL code:

```xml
<dtml-if expr="not butter">
   I can't believe it's not butter.
</dtml-if>
```

is equivalent to:

```xml
<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:

```xml
<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`:
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.

### 7.17.16 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:

```xml
<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="/dtm-var URL"><dtm-var sequence-query
      >query_start=<dtm-var previous-sequence-start-number"></a>  
      (Previous <dtm-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: <dtm-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="/dtm-var URL"><dtm-var sequence-query
      >query_start=<dtm-var next-sequence-start-number"></a>  
      (Next <dtm-var next-sequence-size> results)
  </a>
</dtml-if>

(continues on next page)
```
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>
<p><a href="http://pdx:8090/batch?query_start=11">(Next 10 results)</a></p>
</body>
</html>
```

Another example utilizes the commonly accepted navigation scheme of presenting the the user page numbers from
which to select:

```xml
<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-in>
      b</dtml-var "_['actual_page']"></b>
    </dtml-if>
    <dtml-if sequence-end>
      <dtml-in next-batches mapping>
        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-if>
</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.

### 7.17.17 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 dtm 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:

```xml
<dtml-let lookup="{'a' : 'form15', 'b' : 'form75', 'c' : 'form88'}">
  <dtml-return "_['lookup[REQUEST.get('type')]]">
  </dtml-return>
</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:

```html
<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&nbsp;<a href="<dtml-var URL>?st=logTimea">A</a>&nbsp;<a href="<dtml-var URL>?st=logTimed">D</a></td>
  <td>Type&nbsp;<a href="<dtml-var URL>?st=logTypea">A</a>&nbsp;<a href="<dtml-var URL>?st=logTypeda">D</a></td>
  <td>User&nbsp;<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')[0:-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:

```python
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.

7.17.18 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.

7.18 Searching and Categorizing Content

**Attention:** This document was written for Zope 2.
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, search multiple indexes at once and even specify weighing for different fields in your results. In addition, the **ZCatalog** keeps track of meta-data about indexed objects.

The two most common ZCatalog usage patterns are:

- **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.

### 7.18.1 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 four steps:

- Creating a **ZCatalog**
- Creating indexes
- Finding objects and cataloging them
- Creating a web interface to search the **ZCatalog**

### 7.18.2 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 **Add ZCatalog** form, as shown in the figure below.

![Add ZCatalog form](image)

*Fig. 60: Add ZCatalog 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.

A **ZCatalog** has 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`.

![Add ZCTextIndex Lexicon form](image)

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.
Note: When you want the **TextIndex** to work on other types of objects, they have to provide a method named `PrincipiaSearchSource` which returns the data of the object which has to be searched.

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**, however they should not be confused with the idea of metadata in Zope CMF, Plone, or other content management systems. Here, this just means that these are attributes that will be stored directly in the catalog for performance benefits. 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. Viewing an object in the catalog shows you what was indexed for the object, and what metadata items are stored for it.

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 AnimalCatalog as the searchable object, as shown in the figure below.

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 SearchForm 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, 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. For example, type in the word Reptiles. The AnimalCatalog will be searched and return a simple table.
Add Search Interface

A Search Interface allows you to search Zope databases. The Search Interface will create a search-input form and a report for displaying the search results.

In the form below, *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.

Fig. 63: Creating a search form for a ZCatalog
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.

### 7.18.3 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.

Indexes can be added and removed from a **ZCatalog** using the “pluggable” index interface as shown in the figure below:

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.

Among others, the **ZCatalog** comes with following 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 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.
Fig. 64: Managing indexes

**DateIndex**  A subclass of FieldIndex, optimized for DateTime values. Use this index for any field known to be a date or a DateTime.

**DateRangeIndex**  Searches objects based on a pair of DateTime objects. Use this index to search for objects which are current or in effect at a given time.

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 **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 index and click on the **Delete** button. This will delete the index and all of its indexed content.

**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.

While meta data columns are useful, there are performance tradeoffs from using too many. As more meta data columns are added, the catalog itself becomes larger (and slower), and getting the result objects becomes more memory- and performance-intensive. Therefore, you should choose meta data columns only for those fields that you'll want to show on common search results. Consider carefully before adding a field that returns a large result (like the full text of a document) to meta data.

Next let's look more closely at how to search a ZCatalog.

7.18.4 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 extract 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 meta data. Here is a search form that would allow you to query such a ZCatalog:

```html
<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>
</table>
<input type="submit" name="SUBMIT" value="Submit Query">
</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:
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:

```html
<body tal:define="searchResults context/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>author goes here</td>
      <td>date goes here</td>
    </tr>
  </div>
</table>
</body></html>
```

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 current folder’s 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 Zope Zoo Page Template:

```python
id=context.getId()
return context.NewsCatalog({'contentTextIdx' : id})
```

This template assumes that you have defined absolute_url and title as meta data 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.

**Note:** In reality, you shouldn’t use an index called absolute_url, but should rely instead on the `getURL()` method call below, as that works even in virtual hosting settings.

### Methods of Search Results

The list of results you get for a catalog search is actually a list of Catalog Brain objects. In addition to having an attribute for each item of your meta data, they also have several useful methods:

- **has_key(key)** Returns true if the result object has a meta data element named key.
- **getPath()** Returns the physical path of the result object. This can be used to uniquely identify each object if some kind of post-processing is performed.
- **getURL()** Returns the URL of the result object. You should use this instead of creating a meta data element for absolute_url. This can differ from `getPath()` if you are using virtual hosting.
- **getObject()** Returns the actual Zope object from the result object. This is useful if you want to examine or show an attribute or method of the object that isn’t in the meta data. Once we have the actual object, we can get any normal attribute or method of it. However, be careful not to use this instead of defining meta data. Meta data, being stored in the catalog, is pre-calculated and quickly accessed; getting the same type of information by using `getObject().attribute_name` requires actually pulling your real object from the ZODB and may be a good deal slower. On the other hand, stuffing everything you might ever need into meta data will slow down all querying of your catalog, so you’ll want to strike a balance. A good idea is to list in meta data those things that would normally appear on a tabular search results form; other things that might be needed less commonly (and for fewer result objects at a time) can be retrieved with `getObject`.
- **getRID()** Returns the Catalog’s record id for the result object. This is an implementation detail, and is not useful except for advanced uses.
7.18.5 Searching and Indexing Details

Earlier you saw that the ZCatalog includes several types of indexes. Let’s examine these indexes more closely, and look at some of the additional available indexes, 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:

\[
\text{word1 AND word2}
\]

This will search for all objects that contain both \text{word1} and \text{word2}. Valid boolean operators include \text{AND}, \text{OR}, and \text{NOT}. A synonym for \text{NOT} is a leading hyphen:

\[
\text{word1 -word2}
\]

which would search for occurrences of \text{word1} but would exclude documents which contain \text{word2}. A sequence of words without operators implies \text{AND}. A search for \text{carpet python snakes} translates to \text{carpet AND python AND snakes}.

Parentheses

Control search order with parenthetical expressions:

\[
(\text{word1 AND word2}) \text{ OR word3})
\]

This will return objects containing \text{word1} and \text{word2} or just objects that contain the term \text{word3}.

Wild cards

Search for wild cards like:

\[
Z*
\]

which returns all words that begin with \text{Z}, or:

\[
Zop?
\]

which returns all words that begin with \text{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. \text{?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) AND NOT Zoo*)

will return all objects that contain the terms bob and uncle but will not include any objects that contain words that start with Zoo like Zooologist, 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:

```python
## 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. In addition, stop words, being common words, would appear in almost every page, without this option turned on, a user searching for the python house would get back practically every single document on the site (since they would all likely contain the), taking longer and adding no quality to their results.

**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. *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 FieldIndexes

**FieldIndexes** have different aims than 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 object attributes that contain simple values, such as numbers or short string identifiers.

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
<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:context.NewsCatalog.uniqueValuesFor('author')"
    tal:content="item"
    value="opt value">
</option>
</select>
</td></tr>
<tr><th>Date</th><td><input name="date_index" width=30 value=""></td></tr>
<tr><th>Content</th><td><input name="content_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:
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.

Be careful if you catalog objects with many different values; you can easily end up with a form with a thousand items in the drop-down menu. Also, items must match *exactly*, so strings that differ in capitalization will be considered different.

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 KeywordIndexes

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>
    <option tal:repeat="item python:context.uniqueValuesFor('keywords')">
      opt value goes here
    </option>
  </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:context.AnimalCatalog(pathindex='Zoo/Lizards', meta_type='Image')">
  <a href="url" tal:attributes="href item/getURL 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 a meta data entry for 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 optimized 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 specialized 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.

DateRangeIndexes are widely used in CMF and Plone, where content is compared to an effective date and an expiration date.

DateRangeIndexes also allow one or both of the boundary dates of the indexed objects to be left open which greatly simplifies application logic when querying for active content where expiration and effective dates are optional.

**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.

TopicIndexes are also useful for indexing boolean attributes or attributes where only one value is queried for. They can do this more efficiently then a field index.

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 True. 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 indexed 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:
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.

### 7.18.6 Advanced Searching with Records

A more advanced feature 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.

Note that you don’t have to use record-style queries unless you need the features introduced by them: you can continue to use traditional queries, as demonstrated above.

A record style query involves passing a record (or dictionary) to the catalog instead of a simple query string.

#### KeywordIndex 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:

```python
# 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 FieldIndex (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.
  - **min:max** 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:
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).

Or:

This query matches all animals with population count between 5 and 10 (provided that the same FieldIndex population_count indexing the attribute population_count.)

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:

- '/aa/bb/aa'
- '/aa/bb/bb'
- '/aa/bb/cc'
- '/bb/bb/aa'
- '/bb/bb/bb'
- '/bb/bb/cc'
- '/cc/bb/aa'
- '/cc/bb/bb'
- '/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 behavior 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'
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.

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.

min:max 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:

```python
# return NewsItems newer than a week
zcat = context.NewsCatalog
results = zcat( date={'query' : context.ZopeTime() - 7,
                     'range' : 'min'
                     })
```

('# return NewsItems newer than a week
zcat = context.NewsCatalog
results = zcat( date={'query' : context.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; returning matches where the date supplied to the query falls between the start and end dates from the indexed object.

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:

```html
<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.

7.18.7 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.

If you can trade off memory for time, you can enable Subtransactions in the Advanced tab of the catalog. This commits the work in chunks, reducing memory requirements, but taking longer. It is a good solution for mass cataloging with a very large number of records.

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.
On the other hand, cataloging a complex object when it changes may be too time consuming during operation (especially if the catalog index attempts to translate the information). Some sites may benefit from mass cataloging, and having a cron job or other scheduled job initiate the mass cataloging every night.

In standard (non-CMF, non-Plone) Zope, none of the built-in object types attempt to automatically catalog themselves. In CMF and Plone, the “contentish” object (Documents, News Item, Event, etc.) all use automatic cataloging to add themselves to the standard CMF catalog, `portal_catalog`. CMF and especially Plone offer many advantages; if you’re interested in building a content-oriented site, you should consider these technologies.

### 7.18.8 Advanced Catalog Topics

#### Sorting

When you execute a `ZCatalog` call, your result set may or may not be returned in a particular order:

- If your query contains no text index fields, your results will not be sorted in any particular order. For example, with a query based off a `KeywordIndex`, or query based off both a `KeywordIndex` and a `DateIndex`, you will get an indeterminate ordering.

- For results that include a text index, your results will be returned in order of relevance of the text search. That is, the result set will be sorted based on how often search words appear in the indexes. A search for the word `frog` against a text index will give priority toward an object that uses that word many times compared with an object that uses that fewer. This is a simplified version of the way that many web search engines work: the more relevant your keywords are to an item, the higher its ordering in the results. In particular, with the `ZCTextIndex`, you have a choice between two algorithms for how to weight the sorting:
  
  - Okapi: is the best general choice. It does very well when comparing an ordinary human query against a longer text field. For example, querying a long description field for a short query like `indoor OR mammal` would work very well.
  
  - Cosine: is better suited for when the length of the query comes close to matching the length of the field itself.

You, of course, may want to force a particular order onto your results. You can do this after you get a result set using normal Python syntax:

```python
# get ordered results from search
zcat = context.AnimalCatalog
results = zcat({'title': 'frog'})
results = [(row.title, row) for row in results]
results.sort()
```

This can be, however, very inefficient.

When results are returned by the `ZCatalog`, they are in a special form called a `LazyResults` set. This means that Zope hasn’t gone to the trouble of actually creating the entire list, but has just sketched out the list and will fill it in at the exact point that you ask for each item. This is helpful, since it lets you query the catalog for a result set with 10,000 items without Zope having to really construct a 10,000 item long list of results. However, when we try to sort this, Zope will have to actually create this list since it can’t rely on it’s lazy, just-in-time method.

Normally, you’ll only show the first 20 or 50 or so of a result set, so sorting 10,000 items just to show the first 20 is a waste of time and memory. Instead, we can ask the catalog to do the sorting for us, saving both time and space.

To do this, we’ll pass along several additional keywords in our search method call or query:

- `sort_on` The field name to sort the results on
- `sort_order` ascending or descending, with the default being ascending. Note that you can also use `reverse` as a synonym for descending
**sort_limit** Since you’re likely to only want to use the first 20 or 50 or so items, we can give a hint to the ZCatalog not to bother to sort beyond this by passing along a `sort_limit` parameter, which is the number of records to sort.

For example, assuming we have a `latin_name` FieldIndex on our animals, we can sort them by name in a Python-Script with:

```python
zcat = context.AnimalCatalog
zcat({'sort_on': 'latin_name'})
```

or:

```python
zcat = context.AnimalCatalog
zcat({'sort_on': 'latin_name', 'sort_order': 'descending'})
```

or, if we know we’ll only want to show the first 20 records:

```python
zcat = context.AnimalCatalog
zcat({'sort_on': 'latin_name',
     'sort_order': 'descending',
     'sort_limit': 20})
```

or, combining this with a query restriction:

```python
zcat = context.AnimalCatalog
zcat({'title': 'frog',
     'sort_on': 'latin_name',
     'sort_order': 'descending',
     'sort_limit': 20})[:20]
```

This gives us all records with the **title** frog, sorted by **latin_name**, and doesn’t bother to sort after the first 20 records.

Note that using **sort_limit** does not guarantee that we’ll get exactly that number of records - we may get fewer if they’re aren’t that many matching or query, and we may get more. **sort_limit** is merely a request for optimization. To ensure that we get no more than 20 records, we’ll want to truncate our result set:

```python
zcat = context.AnimalCatalog
zcat({'sort_on': 'latin_name',
     'sort_order': 'descending',
     'sort_limit': 20})[:20]
```

### Unsortable Fields

In order to sort on an index, we have to actually keep the full attribute or method value in that index. For many index types, such as DateIndex or FieldIndex, this is normally done. However, for text indexes, such as ZCTextIndex, the index doesn’t keep the actual attribute or method results in the index. Instead, it cleans up the input (often removing stop words, normalizing input, lowercasing it, removing duplicates, etc., depending on the options chosen. So a term paper with an attribute value of:

"A Critique of 'Tora! Tora! Tora!'"

could actually be indexed as :

```python
('critique', 'tora')
```
once the common stop words (“a”, “of”) are removed, it is lowercased and deduplicated. In reality, the indexed information is much richer, as it keeps track of things like how often words appear, and which words appear earlier in the stream, but this gives you an idea of what is stored.

This is a necessary and positive step to make the index use less storage and less memory, and increases search results, as your site user doesn’t have to worry about getting incidental words (“the”, “a”, etc.) correct, nor about capitalization, etc.

However, this process means that the index no longer knows the actual value, and, therefore, can’t sort on it. Due to this, it is not possible to use the sort_on feature with text indexes types.

To work around this, you can either sort the results of the query using the normal python sort() feature (shown above), or you can create an additional non-text index on the field, described below, in the section Indexing a Field with Two Index Types.

Similarly, the API call uniqueValuesFor, described above, cannot be used on text-type indexes, since the exact values are not kept.

**Searching in More Than One Index Using “OR”**

As mentioned, if you search in more than one index, you must meet your criteria for each index you search in, i.e., there is an implied AND between each of the searches:

```python
# find sunset art by Van Gogh
zcat=context.ArtCatalog
results=zcat({'keyword':'sunsets', 'artist':'Van Gogh'})
```

This query finds all sunset art by Van Gogh: both of these conditions must be true.

There is no way to directly search in more than one index without this AND condition; instead, you can perform two catalog searches and concatenate their results. For example:

```python
# find sunset art OR art by Van Gogh
zcat=context.ArtCatalog
results=zcat({'keyword':'sunsets'}) + \
    zcat({'artist':'Van Gogh'})
```

This method, however, does not remove duplicates, so a painting of a sunset by VanGogh would appear twice.

For an alternate strategy about searching in two places, see PrincipiaSearchSource below.

**Indexing a Field With Two Index Types**

Since the different indexes act differently, it can be advantageous to have the same attribute indexed by more than one index. For example, our animals have a latin_name attribute that gives their formal genus/species latin name. A user should be able to search that trying to match a name exactly, and we should be able to sort results based on that, both of which suggest a FieldIndex. In addition, though, users may want to search that like a text field, where they can match parts of words, in which case we would a ZCTextIndex.

In a case like this, a good strategy is to create one index for the FieldIndex on latin_name. Let’s call that index latin_name. Then, you can create a ZCTextIndex that uses a new feature: the ability to have the indexed attribute be different than the index name itself.

When you create the second index, the ZCTextIndex, you can give it the id latin_name_text, and have the Indexed attributes field be latin_name. Now, when we catalog our animals, their latin_name attribute is indexed in two ways: once, as a FieldIndex, that we can sort against and match exactly, and once as a ZCTextIndex, that we can search like a text field with full text search.
The second index has a different name, so when we make our catalog call, we'll need to be sure to use that name if we want to search it like a text field:

```python
# search latin_name
zcat=context.AnimalCatalog
exact_results=zcat({'latin_name':'homo sapiens'})
fuzzy=zcat({'latin_name_text':'sap*'})
```

Note that a good strategy is to have the search be against the ZCTextIndex, but sort it by the FieldIndex:

```python
# free text search, sorted
zcat=context.AnimalCatalog
results=zcat({'latin_name_text':'sap*','sort_on':'latin_name'})
```

**PrincipiaSearchSource**

You can choose to create indexes on any attribute or method that you would find useful to search on; however, one that is generally helpful is **PrincipiaSearchSource**. Several of the built-in Zope objects, such as DTMLDocuments, and many add-on objects to Zope have a **PrincipiaSearchSource** attribute or method that returns a value that is meant to be used for general purpose searching. Traditionally, **PrincipiaSearchSource** would include the text in an object's title, it's body, and anywhere else you'd want to be able to search.

For example, if you downloaded a Zope product that managed our zoo, and it had an animal type that you could add to your site, this animal type would probably expose a **PrincipiaSearchSource** that looked something like this:

```python
def PrincipiaSearchSource(self):
    """used for general searching for animal""
    return self.title + ' ' + self.latin_name + ' ' \
    + self.description + ' ' + self.environment
```

So that, if you create a **PrincipiaSearchSource** index and search again that, you can find this animal by using words that are in its title, latin_name, description, or environment, without having to worry about which field, exactly, they're in. This is similar to searching with a web search engine, in that you use can use a single text string to find the right information, without needing to know about the type of object you're looking for. It is especially helpful in allowing you to create a site-wide search: searching animals specifically by their latin_name or environment might be useful for a biologist in the right section of your site, but for a general purpose visitor, they might like to search using the phrase jungle and find results without having to know to search for that in the environment field of a search form.

If you create custom types by using more advanced techniques described elsewhere, you should create a **PrincipiaSearchSource** method that returns appropriate object-wide text searching capabilities.

**ZCatalogs and CMF/Plone**

The CMF was built from the ground up to understand the difference between things that are **content**, such as a news item or press release, and those things that are not, such as a **DTMLMethod** used to show a press release, or a **ZCatalog** object. In addition, the CMF includes several stock items that are intended to be used for content, including: Document, Event, NewsItem, and others. These content items are already set up for automatic cataloging, so that any changes made will appear in the catalog.

In non-CMF Zope, the traditional name for a general-purpose catalog is **Catalog**. Though you can always create your own catalog with any id you want; we've used the example **AnimalCatalog** in this chapter for a special-purpose catalog for searching animal-specific info in our zoo. Even though **Catalog** is the traditional name, Zope does not come with such a catalog in the ZODB already, you have to create it.
In CMF (and Plone, an out-of-the-box portal system built on top of the CMF), there is always a catalog created, called `portal_catalog`, at the root of the CMF site. All of the built-in content objects (and almost every add-on content object for the CMF/Plone) are set to automatically catalog to this `portal_catalog`. This is required, since many of the features of the CMF and Plone, such as listing current content, finding content of correct types, etc., rely on the `portal_catalog` and the searching techniques shown here to function.

In CMF and Plone, the index name `PrincipiaSearchSource` is not traditionally used. Instead, an index is created called `SearchableText`, and used in the same manner as `PrincipiaSearchSource`. All of the standard “contentish” objects have a `SearchableText` method that returns things like title, description, body, etc., so that they can be general-text searched.

7.18.9 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.

7.19 Relational Database Connectivity

**Attention:** This document was written for Zope 2.

**Note:** This chapter explains you how to access a relational databases directly through SQL. The alternative and modern way integrating a RDBMS with Zope is using an Object-Relational-Mapper (ORM). An ORM abstracts the SQL layer and allows you to deal with database tables, rows etc. like standard Python objects.

The most common and most flexible ORM in the Python world is SQLAlchemy. You can not use SQLAlchemy directly within Zope because the transaction system of the RDBMS must participate with Zope transaction. This integration layer is implemented through the zope.sqlalchemy module.

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:

<table>
<thead>
<tr>
<th>Row</th>
<th>First Name</th>
<th>Last Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bob</td>
<td>McBob</td>
<td>42</td>
</tr>
<tr>
<td>2</td>
<td>John</td>
<td>Johnson</td>
<td>24</td>
</tr>
<tr>
<td>3</td>
<td>Steve</td>
<td>Smith</td>
<td>38</td>
</tr>
</tbody>
</table>

Information in the table is stored in rows. The table’s column layout is called the *schema*. A standard language, called the Structured Query Language (SQL) is used to query and change tables in relational databases. 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.

### 7.19.1 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.

- **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.

- **SQL Server**  
  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/

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.

### 7.19.2 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 found in the Zope Framework category of the Python Package Index.

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

- **PostgreSQL**  
  The newest and preferred DA is ZPsycopgDA included in psycopg. The older ZpopyDA is also available.

- **MySQL**  
  ZMySQLDA Available as source and a Linux binary package.

- **SQL Server**  
  mxODBC is written by Egenix and very well maintained. There is also ZODBC DA is written by Zope Corporation. Available for the Windows platform only. This DA is no longer actively maintainted.

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.

### 7.19.3 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.

![Fig. 66: PostgreSQL Database Connection](image)

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.

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 [10-3].

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.

### 7.19.4 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. In detail a Z SQL method may contain multiple INSERT or UPDATE statements but at most one SELECT statement.

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.

### 7.19.5 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.

![Z SQL Method Add Form](image)

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:

```sql
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:

```sql
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:

```sql
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.

![Image of Test view](image)

Fig. 70: 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 website.

### 7.19.6 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:

```xml
<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> Euro a year.
      </li>
    </dtml-in>
  </ul>

<dtml-var standard_html_footer>
```

and the ZPT version:

```xml
<div>
  
  <ul>
    <li tal:repeat="row context/list_all_employees">
      <span tal:content="string:${row/id}; ${row/last} ${row/first}
        makes ${row/salary} Euro a year.
      </li>
    </ul>

  </div>
```

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.

### 7.19.7 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`. 

290 Chapter 7. The Zope Book
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:

```xml
<dtm-var standard_html_header>
<dtm-in hire_employee size=50 start=query_start>

<dtm-if sequence-start>

<dtm-if previous-sequence>

<a href="<dtm-var URL><dtm-var sequence-query
    >query_start=<dtm-var
        previous-sequence-start-number"></a>
(Previous <dtm-var previous-sequence-size> results)
</a>

</dtm-if previous-sequence>

<table border>
<tr>
</tr>
</dtm-if sequence-start>

<tr></tr>
</dtm-if sequence-end>

</table>
<dtm-if next-sequence>

<a href="<dtm-var URL><dtm-var sequence-query
    >query_start=<dtm-var
        next-sequence-start-number"></a>
(Next <dtm-var next-sequence-size> results)
</a>

</dtm-if next-sequence>

</dtm-if sequence-end>

<dtm-else sequence-end>

There was no data matching this <dtm-var title_or_id> query.
</dtm-else>

<dtm-in>
<dtm-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:
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.

### 7.19.8 Dynamic SQL Queries

A Z SQL Method query template can contain DTML that is evaluated when the method is called. This DTML can be used to modify the SQL code that is executed by the relational database. Several SQL specific DTML tags exist to assist you in the construction of complex SQL queries. In the next sections you’ll learn about the sqlvar, sqltest and sqlgroup tags.

#### Inserting Arguments with the Sqlvar Tag

It’s pretty important to make sure you insert the right kind of data into a column in a database. You database will complain if you try to use the string “12” where the integer 12 is expected. SQL requires that different types be quoted differently. To make matters worse, different databases have different quoting rules.

In addition to avoiding errors, SQL quoting is important for security. Suppose you had a query that makes a select:

```sql
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:

```sql
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-and>
    </dtml-and>
</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:

```xml
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-and>
      </dtml-sqltest>
    </dtml-sqlgroup>
  <dtml-or>
    <dtml-sqltest salary op="gt" type="float">
    </dtml-sqltest>
  </dtml-sqlgroup>
</dtml-sqlgroup>
```

Given sample arguments, this template renders to SQL like so:

```sql
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.

### 7.19.9 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:
and the ZPT version:

```<div>
    <tal:div tal:repeat="row python: context.employee_by_id(emp_id=42)">
        <h1 tal:content="string: ${row/last}, ${row/first}" />
        <p>
            <span tal:content="string:${row/first}s employee id is ${row/emp_id}. ${row/first} makes ${row/salary} Euro per year.
        </tal:div>
    </div>
```

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:

```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 an organization’s website.

```
  ☐ Departments
    ☐ Engineering
    ☐ Marketing
    ☐ Sales
    ☐ Support

Fig. 71: Folder structure of an organizational website
```

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 four arguments, *department_id*, *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 “requisition-Something”. 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 happen 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:

```html
<dtml-var standard_html_header>
<br />
<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 Scripts (Python) and page templates. For example in Python we could write:

```python
result=context.list_all_employees()
return len(result)
```

and in ZPT:

```html
<span tal:content="python: 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:

```python
[(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:

```python
[{’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:

```html
<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 justBrains. Consider the example Python class:

```python
class Employee:
    def fullName(self):
```

(continues on next page)
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 to 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:

```python
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)

select manager_id from managers where
<dtm-sqltest emp_id type="int" op="eq">
<dtm-in employee_by_id>
    <dtm-var fullName>
</dtm-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:

```sql
select manager_id from managers where
<dtm-sqltest emp_id type="int" op="eq">
<dtm-in employee_by_id>
    <dtm-var fullName>
</dtm-in>
```

You could use this Z SQL Method in your brains class like so:
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:

```python
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]
```

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.

- 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 sites 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.

### 7.19.10 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://mail.zope.org/mailman/listinfo/zope-db](http://mail.zope.org/mailman/listinfo/zope-db)

### 7.19.11 Summary

Zope allows you to build web applications with relational databases. Unlike many web application servers, Zope has its own object database and does not require the use of relational databases to store information.

Zope lets you use relational data just like you use other Zope objects. You can connect your relational data to business logic with scripts and brains, you can query your relational data with Z SQL Methods and presentation tools like DTML, and your can even use advanced Zope features like URL traversal, acquisition, undo and security while working with relational data.
7.20 Virtual Hosting Services

Attention: This document was written for Zope 2.

Zope comes with one object that help you do virtual hosting: Virtual Host Monster. Virtual hosting is a way to serve many websites with one Zope server.

7.20.1 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 ‘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 deciphers 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.

Adding a Virtual Host Monster to your Zope

VirtualHostMonster is one of the add menu items supplied by the stock Zope Product, ‘SiteAccess’. You can add one to any folder by selecting its entry from the add menu and supplying an ID for it (the ID you choose doesn’t matter, except that it must not duplicate the ID of another object in that folder).

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’.
**Configuring the VirtualHostMonster**

The default mode for configuring the VirtualHostMonster is not to do any configuration at all! Rather, the external webserver modifies the request URL to signal what the real public URL for the request is (see “Apache Rewrite Rules” below).

If you do choose to change the settings of your VHM, the easiest method to do so is to use the VHM’s ZMI interface (as explained in the “Virtual Host Monster Mappings Tab” and “Inside-Out Virtual Hosting” sections below).

It is possible to modify the VHM settings from the command line via Zope debugger; no documentation for the low-level API exists, however, except “the source”, ‘Products.SiteAccess.VirtualHostMonster.py, which makes it an inadvisable choice for anyone but an experienced Zope developer.

**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:


  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:


  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 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:
View the DTML Method by clicking on its View tab, and you will see something like the following:

<table>
<thead>
<tr>
<th>Absolute URL</th>
<th><a href="http://localhost:8080/vhm_test">http://localhost:8080/vhm_test</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>URL0</td>
<td><a href="http://localhost:8080/vhm_test/index_html">http://localhost:8080/vhm_test/index_html</a></td>
</tr>
<tr>
<td>URL1</td>
<td><a href="http://localhost:8080/vhm_test">http://localhost:8080/vhm_test</a></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Absolute URL</th>
<th><a href="http://zope.com/vhm_test">http://zope.com/vhm_test</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>URL0</td>
<td><a href="http://zope.com/vhm_test/index_html">http://zope.com/vhm_test/index_html</a></td>
</tr>
<tr>
<td>URL1</td>
<td><a href="http://zope.com/vhm_test">http://zope.com/vhm_test</a></td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Absolute URL</th>
<th><a href="http://zope.com">http://zope.com</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>URL0</td>
<td><a href="http://zope.com/index_html">http://zope.com/index_html</a></td>
</tr>
<tr>
<td>URL1</td>
<td><a href="http://zope.com">http://zope.com</a></td>
</tr>
</tbody>
</table>

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.”

### 7.20.2 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:80/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:

```
www.example.com /path/to/be/rewritten/to
```

You can also match multiple subdomains by putting “*.” in front of the host name in the mapping rule. For example:

```
*.example.com /folder
```

This example will match “my.example.com”, “zoom.example.com”, etc. If an exact match exists, it is used instead of a wildcard match.

The best way to explain how to use the Mappings tab is by more specific 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’. You will see:

```
Absolute URL http://www.example.com:8080
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’. Note that it is not possible to rewrite the port part (by default, ‘8080’) of the URL this way. To change the port Zope is listening on, you will have to configure Zope’s start parameter or use Apache rewriting.

**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 apply the contained directives to the specified virtual host.

Using Apache’s rewrite rule functionality requires that the ‘mod_rewrite’ and ‘mod_proxy’ Apache modules be enabled. This can for instance be done by configuring Apache with the ‘–enable-modules=”rewrite proxy”’ flag during compile time or by loading the corresponding shared modules.

If you are using the new Apache 2 series, you will also have to include the ‘mod_proxy_http’ module. See the “Apache mod_rewrite documentation”, http://httpd.apache.org/docs/trunk/mod/mod_rewrite.html for details.

You can check whether you have the required modules installed in Apache by examining ‘LoadModule’ section of httpd.conf.

After you’ve got Apache configured with mod_rewrite and mod_proxy (and, depending on your Apache version, mod_proxy_http), 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:WINNTsystem32driversetchosts 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.

Note: On MacOS X Server, the ‘Server Admin.app’ program simplifies adding virtual host definitions to your Apache. This application can make and maintain virtual host, access log, etc.

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 *:80
<VirtualHost *:80>
  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>
```

If you want to proxy SSL to Zope, you need a similar directive for port 443:

```
NameVirtualHost *:443
<VirtualHost *:443>
  ServerName www.example.com
  SSLProxyEngine on
  RewriteEngine On
  RewriteRule ^/(.*) http://127.0.0.1:8080/VirtualHostBase/https/www.example.com:443/vhm_test/VirtualHostRoot/$1 [L,P]
</VirtualHost>
```

Note: the long lines in the RewriteRule directive above must remain on a single line, in order for Apache’s configuration parser to accept it.

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”, http://httpd.apache.org/docs/2.0/misc/rewriteguide.html for more information on the subject of rewrite rules.

7.20.3 Virtual Hosting Considerations for Content classes

Be sure that content objects catalog themselves using as their unique ID a “site-relative” path, rather than their full physical path; otherwise, the object will be findable when using the site without virtual hosting, but not with, or vice versa.

7.20.4 “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.

7.21 Session Management

Attention: This document was written for Zope 2.

This chapter describes Zope’s built-in Session Management.

7.21.1 Terminology

Here’s a mini-glossary of of key terms used within this document:

Web session a series of HTTP requests from the same browser to the same server during that browser’s execution life-span.

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-.

7.21.2 Session Managers

Web browsers communicate with Web Servers using HTTP. HTTP does not provide tools that can track users and data in the context of a web session. Zope’s session management works around the problem: it provides methods able to track site visitor activity. Applications like “shopping carts” use session management for this reason.

Zope’s session management makes use of name-spaces like cookies, HTTP form elements, and/or parts of URLs “in the background” to keep track of user sessions. Which of these name-spaces are used is configurable using the browser_id manager (described later).

Session data is valid for the duration of a configurable inactivity timeout value or browser shut-down, which ever comes first. Zope’s session management keeps track of anonymous users as well as those who have Zope login accounts.

Important! Data maintained by Zope’s session management is no more secure than HTTP itself. A session is secure if and only if:

- the connection between a browser and Zope uses strong encryption (SSL normally).
- precautions specific to the security exposure are taken.

It’s clear that you should not store sensitive information like credit card numbers in a session container unless you understand the vulnerabilities. See the section entitled Security Considerations near the end of this document.

It is advisable to use sessions only on pages where they are necessary because of a performance impact on your application. The severity varies depending on usage and configuration. A good “rule of thumb” is to account for a 5% - 10% speed-of-execution penalty.

Some hints:

- Do not use SESSION to store REQUEST variables. They are already available in the REQUEST.
- Do not store any data in SESSION that you can get from the Zope API. Its faster (and more secure) to get user Id from Zope’s Security Manager then it is from the SESSION object.

7.21.3 Session Manager Components

Browser Id Manager

This component 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 Transient Data Object (REQUEST.SESSION) which delegates 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.
- The default sessioning configuration provides a Browser Id Manager as the:
Session Data Manager

This component 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;
- creates a new session data object or hands back an existing session data object based on the browser id.
- Developers generally do not directly use methods of session data managers to obtain session data objects. Instead, they 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 has an identifier distinct from 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).
- The default sessioning configuration provides a Session Data Manager named:

```
/session_data_manager
```

Transient Object Container

Also known as Session Data Containers, these components actually hold information related to sessions.

- Currently, a Transient Object Container is used to hold a special “transient 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.
- 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.

Transient Data Object

Also known as the 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.
7.21.4 Using Session Data

You will typically access session data through the:

```python
SESSION
```

attribute of the REQUEST object. Session data objects are like Python dictionaries: they can hold almost any kind of object as a key or a value. It’s likely you will almost always use “normal” Python objects such as lists, dictionaries, strings, and numbers.

Here’s an example of how to work with a session using a Python Script:

```python
## 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'
    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'
session['last view']=context.ZopeTime()
return 'This is your first view'
```

This example shows how to access SESSION data. But it is not a “best practice” example. If performance is an issue, you should not attempt to keep 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 then click its Test tab. While viewing the output, reload the 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 were ever there. 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.

See the Concepts and Caveats section at the end of this document for things to watch out for while accessing Zope’s Session Manager “naively”.

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):

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

Sessions have additional configuration parameters and usage patterns detailed below.

7.21.5 Default Configuration

Zope is preconfigured with a default sessioning setup.

The Zope “default” browser id manager lives in the root folder and is named:

```python
browser_id_manager
```

The Zope “default” session data manager lives in the root folder and is named:

```python
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.

### 7.21.6 Advanced Development Using Sessioning

#### Overview

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 with 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.

#### 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:

```python
goingSessionData()
```

method. In ZPT:

```python
dlta python:context.session_data_manager.getSessionData(create=0)"
```

Note: create=0 means return a reference to the session or None. create=1 means return a reference if one exists or create a new Session object and the reference.

### Modifying A Session Data Object

Once you’ve used REQUEST.SESSION or:

```python
session_data_manager.getSessionData()
```

to obtain a session data object, you can set key/value pairs of that session data object.

In ZPT:

```python
dlta python: request.SESSION">
    <tal:block define="temp python: data.set('foo','bar')">
        <p tal:content="python: data.get('foo')">bar will print here</p>
    </tal:block>
</span>
```

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 for 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.

### Clearing A Session Data Object

You can clear all keys and values from a SESSION object by simply calling its clear() method.

In ZPT:

```python
dlta python: request.SESSION.clear()"
```

### 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.

There is a caveat. If you invalidate the session object in a script then you **must** obtain a fresh copy of the session object by calling getSessionData and not by reference (REQUEST.SESSION).

Here is an example using DTML::
Manual invalidation of session data is useful when you need a “fresh” copy of a session data object. 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.

**Using Session Data with TAL**

Here’s an example of using the session data object with TAL:

```
<span tal:define="a python:request.SESSION;
    dummy python:a.set('zopetime',context.ZopeTime())">
    <p tal:content="python: a.get('zopetime')"></p>
</span>
```

**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 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:

<dtm-var "REQUEST.SESSION.getBrowserIdManager().getBrowserId()">

Another way of doing this, which returns the same value is:

<dtm-var "REQUEST.SESSION.getContainerKey()">

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:

<dtm-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 ("cookies", "form", or "url") the browser id has been gathered.

It should be noted that you can configure the browser_id_manager (its in Zope root by default) so that it searches whatever combination of namespaces you select.

There are three methods of browser id managers which allow you to accomplish this:

<dtm-if "REQUEST.SESSION.getBrowserIdManager().isBrowserIdFromCookie()">
    The browser id came from a cookie.
</dtm-if>

<dtm-if "REQUEST.SESSION.getBrowserIdManager().isBrowserIdFromForm()">
    The browser id came from a form.
</dtm-if>

(continues on next page)
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. Here’s a TAL example:

```xml
<span tal:define="idManager python:request.SESSION.getBrowserIdManager()">
  <form action="thenextmethod">
    <input type=submit name="submit" value=" GO ">
    <input type="hidden" name="name" value="value"
      tal:attributes="name python: idManager.getBrowserIdName();
                      value python: idManager.getBrowserId()">
  </form>
</span>
```

A convenience function exists for performing this action as a method of a browser id manager named “getHiddenFormField”:
When the above 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.

Using formvar-based sessioning.

To use formvar-based sessioning, you need to encode a link to its URL with the browser id by using the browser id manager’s:

```
encodeUrl()
```

method.

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:
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:

```python
hasSessionData()
```

method of the session data manager. This method returns true if there is session data associated with the current browser id:

```html
<dtm1-if "session_data_manager.hasSessionData()">
    The sessiondatamanager object has session data for the browser id associated with this request.
</dtm1-if>
```

The:

```python
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:

```python
encodeUrl()
```

method of browser id managers in order to perform this encoding:

```html
<html>
<body>
    <a href="<dtm1-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
<html>
<body>
  <a href="/amethod?_ZopeId=7HJhy78978979JHK">Here</a>

  <a href="/amethod?_ZopeId=7HJhy78978979JHK">Here</a>

  is a link.
</body>
</html>
```

You may successfully pass URLs which already contain query strings to the:

```python
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:

```python
URLs
```

in the “Look for Browser Id in” form element of your browser id manager):

```html
<html>
<body>
  <a href="dtml-var "REQUEST.SESSION.getBrowserIdManager().encodeUrl('/amethod',
˓
→style='inline')">">Here</a>

  <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
<html>
<body>
  <a href="/ZopeId/7HJhy78978979JHK/amethod">Here</a>

  <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. You might want an onAdd method but not an onDelete method. You may define one, both or none of the TOC event methods.

Here are examples of the kinds of things Session onAdd and onDelete methods are used to do:

- The onAdd method can be used to populate a session data object with “default” values before it’s used by application code.
- The onDelete method can write the contents of a session data object out to a permanent data store before it is timed out or invalidated.
You can manually configure the onAdd and onDelete methods. Click the management tab of ‘temp_folder\session_data’. Enter “a physical path” to either an external method or python script. NOTE: This configuration is only good until the next Zope shutdown because:

\temp_folder\session_data

is in a RAM database, Configure the onAdd and onDelete methods for this data container via the:

zope.conf

configuration file for your Zope instance. This is covered in some detail in Setting Initial Transient Object Container Parameters later in this document.

Note: the 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. In recent versions of Zope, the event.log defaults to Zope-Instance/log/event.log. This is configurable in:

zope.conf

### 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 can be written to serve the purpose of being called on session data object creation and invalidation.

The Script (Python) 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:

```python
onAdd
```

which had function parameters “sdo” and “toc” and a body of:

```python
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:

```python
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:

```python
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.

7.21.7 Configuration and Operation

Setting the default Transient Object Container Parameters

Click on:

```
/temp_folder/session_data
```

and you’ll see options to control inactivity time-outs and the maximum allowable number of Session objects. You can even include paths to python scripts that handle a Session’s after-add and before-delete events.

Because:

```
/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 any changes to parameters will be lost the next time you restart your Zope server.

If you need to permanently alter the default Transient Object Container’s configuration you must edit Zope’s startup configuration file:

```
zope.conf
```

Note that additional Transient Object Containers can be instantiated in permanent storage. They are rarely needed. If you do need this its covered in detail later in this document.

Here is the relevant portion of zope.conf:

```bash
# Directive: maximum-number-of-session-objects
# Description: An integer value representing the maximum number
#   of subobjects"
# allowable in the '/temp_folder/session_data' transient object container.
#
# Default: 1000
# Example: maximum-number-of-session-objects 10000
```

(continues on next page)
# Directive: session-add-notify-script-path
#
# Description:
# An optional fill 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 the /temp_folder folder at startup.
#
# Default: unset
# Example: session-add-notify-script-path /scripts/add_notifier
#
# Directive: session-delete-notify-script-path
#
# Description:
# An optional fill 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 the /temp_folder folder at startup.
#
# Default: unset
# Example: session-delete-notify-script-path /scripts/del_notifier
#
# Directive: session-timeout-minutes
#
# Description:
# An integer value representing the number of minutes to be used as the
# "data object timeout" of the '/temp_folder/session_data' transient
# object container.
#
# Default: 20
# Example: session-timeout-minutes 30
#
# Directive: session-resolution-seconds
#
# Description:
# An integer value representing the number of seconds to be used as the
# "timeout resolution" of the '/temp_folder/session_data' transient
# object container.
#
# Default: 20
# Example: session-resolution-seconds 60

Instantiating Multiple Browser Id Managers (Optional)

Transient data objects depend on a session data manager, which in turn depends on a browser id manager. A browser id manager 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.

In the container of your choosing, select “Browser Id Manager” from the add drop-down 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.

**Cookie Domain** this is the “domain” element which should be sent in the browser id cookie. 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
```

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” drop-down 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. Note: session manager’s should not share transient object paths. This is an
example path:

Zope transient object container is:

```
/MyTransientSessionFolder
```

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 and their data disappear when you restart Zope.

If you want your session data to persist across server reboots, or if you have a 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. You can “mount” additional storages within the `zope.conf` file of your Zope instance.
The default `temp_folder` is mounted inside a `TemporaryStorage`, which is nonundoing and RAM-based. There are
other nonundoing storages, such as BerkeleyStorage, although none quite as well-supported as TemporaryStorage.

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
to 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`. 

7.21. Session Management
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 ObjectExpiration Considerations” in the *Concepts and Caveats* section below for details on session data expiration.

**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.

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.
### 7.21.8 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

A browser id will last 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.

The transient object container 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 a 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:

```python
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:

```python
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
session.invalidate() and stale references to the session object

This Python Script illustrates an issue with using the invalidate method of a session object:

request = container.REQUEST
session = request.SESSION
session.set('foo','bar')
session.invalidate()
# ............................................
# we expect that invalidate() flushes the session
# ............................................
print 'after invalidate()',session.get('foo') # 'bar' still prints!

# ............................................
# Even this isn't enough
# ............................................
session = request.SESSION
print 'after invalidate()', session.get('foo') # 'bar' still prints!

# ............................................
# Here's the work-around
# ............................................
session = context.session_data_manager.getSessionData()
print 'after getSessionData', session.get('foo') # 'bar' is GONE which is good
return printed

In short, after using the `invalidate` method of a session object, the next reference to the session object you obtain should be through “getSessionData” rather than `REQUEST.SESSION`.

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](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

Mounted TOC’s do not acquire parameter’s from zope.conf (which is the case for the default transient object container). Therefore you set parameters directly on the object in ZMI.

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
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:

```
2009-01-16T04:26:58 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.

### 7.22 Scalability and ZEO

**Attention:** This document was written for Zope 2.

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. ZEO is built into Zope, no additional software install is required.

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 ZODB and ZEO discussion area.

7.22.1 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.

![Simple ZEO illustration](image)

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, could distribute a Zope site to disparate geographic locations, given good network connectivity between the ZEO clients and the ZEO server. In this chapter we’ll explore some interesting ways you can distribute your ZEO clients.
7.22.2 When you should use ZEO

Using a ZEO-based installation is advantageous for almost all users. Here are some of the reasons:

- 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. ZEO enables the developer to attach to a ZODB database while still continuing to serve requests from another ZEO client.

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.

7.22.3 Installing and Running ZEO

ZEO is part of Zope, all batteries are included. However, 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. The easiest way to meet this prerequisite is to make sure all of your computers use the same Zope version.

- 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 is very easy. After you have gone through the steps necessary to build the Zope software it takes nothing more than running two scripts and tweaking the default configuration laid down in the ZEO client’s `zeo.conf` configuration file.

First, you need to create a place where the ZEO server will live. It also contains the database file, so make sure you have enough space to cover your expected database size and at least double that so you can pack the ZODB:

```
$ python /path/to/Zope/bin/mkzeoinstance.py /path/to/zeostorage
```

Make sure you use the same python interpreter that was used to build your Zope software. `/path/to/zeostorage` represents the location where you want the ZEO server to be. While the script runs you will see output telling you what it is doing.

Once you have built the ZEO server’s home this way you will notice that its layout is very similar to a Zope instance home. It has a configuration file named `zeo.conf` inside its etc-subdirectory which you should look at to get a notion of what can be configured, and you will need it to look up where the server will listen for ZEO requests when you configure your ZEO clients.

The ZEO storage home also contains prefabricated start/stop scripts that work the same way as the Zope `zopectl` script, for ZEO it is called `zeoctl`. 
You should now have ZEO properly installed. Try it out by first starting the server. In a terminal window or DOS box type:

```bash
$ /path/to/zeostorage/bin/zeoctl start
```

You can follow its log file by simply typing:

```bash
$ /path/to/zeostorage/bin/zeoctl logtail
```

or by looking at the log file directly. Its location is configurable using the previously mentioned zeo.conf configuration file.

After having set up the ZEO storage server that way you will want at least one ZEO client. You can use an existing Zope server (provided it meets the prerequisites mentioned earlier) or build a new instance home the same way you would if you set up a new Zope server without ZEO:

```bash
$ python /path/to/Zope/bin/mkzopeinstance
```

Now visit the instance home you created and look for the `zope.conf` configuration file in its etc-directory. In order to use ZEO the client must be told to access the ZODB not from the file system but talk to a ZEO server instead. Look for the:

```ini
zodb_db main
```

directive at the bottom. Underneath the default configuration you will notice an example ZEO client configuration. Comment out the complete zodb_db main stanza containing the:

```ini
filestorage
```
directive and uncomment the example zodb_db main configuration that contains the:

```ini
zeoclient
```
directive. If you have not tweaked your zeo.conf file all you need to do at this moment is to ensure that the `server` argument in the `zeoclient` directive shows the same value as the `address` argument in the `zeo` directive inside your ZEO server's zeo.conf file.

Now you are ready to test the ZEO client. Fire it up by running:

```bash
$ /path/to/zeoclient/bin/zopectl start
```

and check the log file manually or by running:

```bash
$ /path/to/zeoclient/bin/zopectl logtail
```

Now visit the Zope Managment Interface (ZMI) of your ZEO client in a web browser and go to the Control Panel. 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. This can easily be achieved by creating more ZEO client instances as described above. The instances can be on the same server machine or distributed over several machines.

### 7.22.4 How to Distribute Load

Imagine 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.

Now, add some simple TAL code to your site to display a list of your mirrors:

```python
<h2>Please choose from the following mirrors: <ul>
   <li tal:repeat="mirror here/mirrors">
      <a href="" tal:attributes="href mirror" tal:content="mirror">
         my.mirror.site
      </a>
   </li>
</ul>

Or, in a Script (Python):

```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:
```
This TAL code (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:

```
<tal:block define="mirror python: modules.random.choice(here.mirrors);
dummy python: request.RESPONSE.redirect(mirror)"/>
```

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.

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.
Fig. 75: Load balancing with round-robin DNS
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.

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 website.

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.

Other software-based solutions

If you are looking for a simple load balancer and proxy software to put in front of your ZEO clients you can take a look at the Pound load balancer which can be set up quickly and offers many convenient features.

Many administrators will want to cache content and load balance at the same time. The Squid cache server is an excellent choice. Toby Dickenson has written up a HowTo describing a configuration in which Squid caches and balances the load among several ZEO clients.

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
Fig. 76: Illustration of Layer 4 switching
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 storage works 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.

![Zope connected to a filestorage](image)

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.

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).
Fig. 78: Zope with a Client Storage and Storage server
7.22.5 Ongoing Maintenance

A ZEO server does not need much in terms of care and feeding. You need to make sure the ZODB does not grow too large and pack it once in a while, and you should rotate the server logs.

Packing

FileStorage, the most common ZODB database format, works by appending changes at the file end. That means it will grow with time. To avoid running out of space it can be packed, a process that will remove old object revisions and shrink the ZODB. Zope comes with a handy utility script to do this task, and you can run it in an automated fashion like out of cron. Look for a script named zeopack.py underneath ZODBTools in the utilities directory of your Zope installation.

Given a setup where the ZEO server is listening on port 8001 on localhost, you pack it this way:

```
$ python /path/to/Zope/utilities/ZODBTools/zeopack.py -h localhost -p 8001
```

Make sure you use the same version of Python that is used to run the ZEO server.

Log Rotation

ZEO by default keeps a single event log. It is located in the log subdirectory of your ZEO server’s home and can be configured using the zeo.conf configuration file. Depending on the level of logging specified and server traffic the file can grow quite quickly.

The zeoctl script in your ZEO storage home has a facility to effect the closing and reopening of the log file. All you need to do is move the old log aside and tell the server to start a new one:

```
$ cd /path/to/zeostorage
$ mv logs/zeo.log logs/zeo.log.1
$ bin/zeoctl logreopen
```

These steps can be automated via cron, at on Windows or the handy logrotate facility on Linux. Here is an example logrotate script that can be dropped into ‘/etc/logrotate.d’:

```
# Rotate ZEO logs weekly
/path/to/zeostorage/log/zeo.log { weekly
  rotate 5
  compress
 notifempty
  missingok
  postrotate
  /path/to/zeostorage/bin/zeoctl logreopen
  endscript
}
```

7.22.6 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.

ZEO servers do not have any in-memory cache for frequently or recently accessed items. Every request for an object from a ZEO client will cause a read from disk. While some of that read activity is served by operating system level disk caches or hardware caches built into the drive itself it can still make the server quite busy if multiple ZEO clients are in use. It is good practice to ensure that a busy ZEO server has a fast disk.

To maximize serving speed for ZEO clients (which necessitates minimizing trips to the ZEO server for retrieving content) it is advisable to keep a large ZEO client cache. This cache keeps frequently accessed objects in memory on the ZEO client. The cache size is set inside the zeoclient stanza in the zodb_db main section of your ZEO client’s zope.conf file. Using the key cache-size you can specify an integer value for the number of bytes used as the ZEO cache. By default this is set to a value of 20000000, which equates about 20 MB. Zope allows you to use a simpler format such as 256MB for the cache-size key.

7.22.7 Conclusion

In this chapter we looked at ZEO, and how ZEO can substantially increase 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 website. This complexity pays off however when you need to serve up lots of dynamic content to your audience.

7.23 Managing Zope Objects Using External Tools

Attention: This document was written for Zope 2.

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 auto-indent, no key re-bindings, 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 feature-rich 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.

### 7.23.1 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 extension-less “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, the semantics of Zope can interfere with the experience. The same file on your hard drive, 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.

### 7.23.2 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.

Some examples of editors and applications that support FTP are Homesite, KDE suite of applications (Kate, Quanta, Kwrite, Konqueror), Bluefish, and Dreamweaver.

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.

7.23.3 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 editing the necessary incantation in your `INSTANCE_HOME/etc/zope.conf` 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.

To transfer objects between your Zope server and local computer:

- start WS_FTP and enter the Zope IP address or machine name and port information.
- Click the “Connect” button.
- 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.

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.

**Transferring files with KDE’s Konqueror**

KDE is one of the many popular window manager for Unix. KDE comes with many applications that is FTP enabled. One such application is Konqueror. Konqueror is a file manager, and also works as a browser.

To use Konqueror to transfer files to your zope site:

- enter `ftp://username@your.server.com:port`
- Enter your username and password when prompted.

Once the correct password is presented, you can now transfer files to and from your zope site.

With Konqueror, you can split the Konqueror view, and make it to mimic WS_FTP, or Midnight Commander (a popular menu based file manager), as shown in the figure below.

---

**Fig. 79: 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.

**Transferring files with KDE’s Konqueror**

KDE is one of the many popular window manager for Unix. KDE comes with many applications that is FTP enabled. One such application is Konqueror. Konqueror is a file manager, and also works as a browser.

To use Konqueror to transfer files to your zope site:

- enter `ftp://username@your.server.com:port`
- Enter your username and password when prompted.

Once the correct password is presented, you can now transfer files to and from your zope site.

With Konqueror, you can split the Konqueror view, and make it to mimic WS_FTP, or Midnight Commander (a popular menu based file manager), as shown in the figure below.

---

7.23. Managing Zope Objects Using External Tools 345
Fig. 80: Viewing the Zope object hierarchy with Konqueror
We can also edit, create or delete some known Zope objects like folder or ZPT. For instance, to edit a file-like object, right click > Open With > Choose Application > Kate. You can start editing away. Kate will do the necessary when you save your edits.

**Transferring files with MS Internet Explorer 6+**

MS Internet Explorer version 6 and above can also do FTP. To use MS Internet Explorer to move files between your desktop and Zope:

- enter ftp://your.server.com:port
- click “File” > “Login as”.
- Enter your username and password when prompted.

You can then create new Folders and transfer files between Zope and your desktop, as shown in the figure below.

![Fig. 81: Viewing the Zope object hierarchy with IE](image)

**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:

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:

```lisp
(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" "")")
```

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 KDE Desktop

KDE comes with many applications that is FTP aware. For example, Kate, Kwrite, Quanta, Konqueror, and many more.

To start editing objects with Kate:

- Click “File” > “Open”.
- Enter the location “ftp://user@server:port/”
Fig. 82: Viewing the Zope Root Folder via ange-ftp
• Browse and select the zope object you want to edit.

Once selected, you can edit to your heart’s content, and click “File” > “Save” when done. Kate will save your edit to your zope server.

![Open File - Kate](image)

Fig. 83: Viewing the Zope Root Folder via Kate/KDE desktop

With KDE, you can also mount zope onto your dialog box. To do that:

• click “File” > “Open”.
• Right click on the listed locations in the “Open” dialog box
• Click “Add Entry”.
• Fill in “Zope ftp” or any other description in the description field.
• Enter the URL “ftp://user@server:port/” in the location field.
• Select your icon.

Now, you can edit zope objects in a single click.

### 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 websites. 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. To enable Zope’s WebDAV source server, enter the following in zope.conf:

```xml
<webdav-source-server>
  # valid keys are "address" and "force-connection-close"
  address 8022
  force-connection-close off
</webdav-source-server>
```

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 for 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 be confronted.
Zope 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.

Editing Zope objects with cadaver

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: ZopeBook 0 Jul 22:57
Coll: temp_folder 0 Jul 19:47
Coll: tutorial 0 Jun 00:42
acl_users 0 Dec 2009
browser_id_manager 0 Jun 14:01
index_html 93 Jul 01:01
session_data_manager 0 Jun 14:01
standard_error_message 1365 Jan 2009
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 sitecopy program can inspect your local and remote data and only transfer the changes, using FTP or DAV.
Editing Zope objects with KDE applications

KDE applications are WebDAV aware. Therefore, we can actually edit Zope objects from any of the KDE applications, such as konqueror, quanta, kate, et cetera.

Using konqueror:

- enter:
  \[\text{webdav://your.server:port/}\] in the konqueror location.

- enter the username and password when prompted.
- start editing when konqueror presents the Zope workspace.

Using Kate:

- Open Kate
- Click File > Open
- Enter:
  \[\text{webdav://your.server:port/}\]

in “Open File dialog” “Location”

Fig. 85: Viewing the Zope object hierarchy with konquerorWebDAV
• Browse for your file or start editing.

Fig. 86: Kate Open File dialog box WebDAV

7.23.4 Other Integration Facilities

This chapter focused on FTP and DAV. 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.

7.24 Maintaining Zope

Attention: This document was written for Zope 2.

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
7.24.1 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). 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 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.

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 $ZOPE_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.

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:
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.

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 located in:

```
/etc/rc.d/init.d/zope
```

if your system puts the:

```
init.d
```
directory somewhere else, you should accomodate 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:
on shutdown.

A simple generic startup script structure could be something like this:

```bash
#!/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
MS Windows

The prevalent way to autostart Zope on MS Windows is to install Zope as a service.

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.

7.24.2 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:
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
```

### 7.24.3 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.

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.

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:

```python
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.

7.24.4 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 ‘$ZOPE_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 $ZOPE_HOME/var/Z2.pid`
```

**SIGTERM** close open database connections then shut down. The common idiom for shutting down Zope is:

```
kill -TERM `cat $ZOPE_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 $ZOPE_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.

### 7.24.5 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.
```

(continues on next page)
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 Zope's logging system.

You also should look at your access log (usually placed in $ZOPE_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):

```bash
#!/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.

**7.24.6 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 $ZOPE_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 messages. The ordinary log method is to log to a file specified by the EVENT_LOG_FILE, eg. EVENT_LOG_FILE=$ZOPE_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:

```bash
compress
/usr/local/zope/var/Z2.log {
rotate 25
weekly
postrotate
/sbin/kill -USR2 `cat /usr/local/zope/var/Z2.pid`
endscript}
```

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.

**7.24.7 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 Zope’s 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.

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 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.

### 7.24.8 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
  - **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.
```
7.25 Appendix A: DTML Reference

Attention: This document was written for Zope 2.

DTML is the Document Template Markup Language, a handy presentation and templating language that comes with Zope. This Appendix is a reference to all of DTMLs markup tags and how they work.

7.25.1 call: Call a method

The ‘call’ tag lets you call a method without inserting the results into the DTML output.

Syntax

‘call’ tag syntax:

```
<dtml-call Variable|expr="Expression"
```

If the call tag uses a variable, the methods arguments are passed automatically by DTML just as with the ‘var’ tag. If the method is specified in a expression, then you must pass the arguments yourself.

Examples

Calling by variable name:

```
<dtml-call UpdateInfo
```

This calls the ‘UpdateInfo’ object automatically passing arguments.

Calling by expression:

```
<dtml-call expr="RESPONSE.setHeader('content-type', 'text/plain')"
```

See Also

• var tag

7.25.2 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:

```html
<dtml-comment>
This content is not executed and does not appear in the output.
</dtml-comment>
```

Commenting out DTML:

```html
<dtml-comment>
This DTML is disabled and will not be executed.
<dtml-call someMethod>
</dtml-comment>
```

Zope still validates the DTML inside the comment block and will not save any comments that are not valid DTML. It is also not possible to comment in a way that breaks code flow, for example you cannot improperly nest a comment and a dtml-in.

**7.25.3 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, “fooobar”)’ is equivalent to ‘x.fooobar’. If the named attribute does not exist, default is returned if provided, otherwise ‘AttributeError’ is raised.

- **getitem(variable, render=0)** Returns the value of a DTML variable. If ‘render’ is true, the variable is rendered. See the ‘render’ function.
hasattr(object, string)  The arguments are an object and a string. The result is 1 if the string is the name of one of the object’s attributes, 0 if not. (This is implemented by calling getattr(object, name) and seeing whether it raises an exception or not.)

hash(object)  Return the hash value of the object (if it has one). Hash values are integers. They are used to quickly compare dictionary keys during a dictionary lookup. Numeric values that compare equal have the same hash value (even if they are of different types, e.g. 1 and 1.0).

has_key(variable)  Returns true if the DTML namespace contains the named variable.

hex(integer)  Convert an integer number (of any size) to a hexadecimal string. The result is a valid Python expression. Note: this always yields an unsigned literal, e.g. on a 32-bit machine, ‘hex(-1)’ yields ‘0xffffffff’. When evaluated on a machine with the same word size, this literal is evaluated as -1; at a different word size, it may turn up as a large positive number or raise an ‘OverflowError’ exception.

int(number)  Convert a string or number to a plain integer. If the argument is a string, it must contain a possibly signed decimal number representable as a Python integer, possibly embedded in whitespace; this behaves identical to ‘string.atoi(number[, radix])’. The ‘radix’ parameter gives the base for the conversion and may be any integer in the range 2 to 36. If ‘radix’ is specified and the number is not a string, ‘TypeError’ is raised. Otherwise, the argument may be a plain or long integer or a floating point number. Conversion of floating point numbers to integers is defined by the C semantics; normally the conversion truncates towards zero.

len(sequence)  Return the length (the number of items) of an object. The argument may be a sequence (string, tuple or list) or a mapping (dictionary).

max(s)  With a single argument s, return the largest item of a non-empty sequence (e.g., a string, tuple or list). With more than one argument, return the largest of the arguments.

min(s)  With a single argument s, return the smallest item of a non-empty sequence (e.g., a string, tuple or list). With more than one argument, return the smallest of the arguments.

namespace([name=value]...)  Returns a new DTML namespace object. Keyword argument ‘name=value’ pairs are pushed into the new namespace.

oct(integer)  Convert an integer number (of any size) to an octal string. The result is a valid Python expression. Note: this always yields an unsigned literal, e.g. on a 32-bit machine, ‘oct(-1)’ yields ‘037777777777’. When evaluated on a machine with the same word size, this literal is evaluated as -1; at a different word size, it may turn up as a large positive number or raise an OverflowError exception.

ord(character)  Return the ASCII value of a string of one character. E.g., ‘ord(“a”)’ returns the integer 97. This is the inverse of ‘chr()’.

pow(x, y [,z])  Return x to the power y; if z is present, return x to the power y, modulo z (computed more efficiently than ‘pow(x, y) % z’). The arguments must have numeric types. With mixed operand types, the rules for binary arithmetic operators apply. The effective operand type is also the type of the result; if the result is not expressible in this type, the function raises an exception; e.g., ‘pow(2, -1)’ or ‘pow(2, 35000)’ is not allowed.

range([start,] stop [,step])  This is a versatile function to create lists containing arithmetic progressions. The arguments must be plain integers. If the step argument is omitted, it defaults to 1. If the start argument is omitted, it defaults to 0. The full form returns a list of plain integers ‘[start, start + step, start + 2 * step, ...]’. If step is positive, the last element is the largest ‘start + i * step’ less than stop; if step is negative, the last element is the largest ‘start + i * step’ greater than stop. step must not be zero (or else ‘ValueError’ is raised).

round(x [,n])  Return the floating point value x rounded to n digits after the decimal point. If n is omitted, it defaults to zero. The result is a floating point number. Values are rounded to the closest multiple of 10 to the power minus n; if two multiples are equally close, rounding is done away from 0 (so e.g. round(0.5) is 1.0 and round(-0.5) is -1.0).

render(object)  Render ‘object’. For DTML objects this evaluates the DTML code with the current namespace. For other objects, this is equivalent to ‘str(object)’.
reorder(s, [with], [without]) Reorder the items in s according to the order given in ‘with’ and without the items mentioned in ‘without’. Items from s not mentioned in with are removed. s, with, and without are all either sequences of strings or sequences of key-value tuples, with ordering done on the keys. This function is useful for constructing ordered select lists.

SecurityCalledByExecutable() Return a true if the current object (e.g. DTML document or method) is being called by an executable (e.g. another DTML document or method, a script or a SQL method).

SecurityCheckPermission(permission, object) Check whether the security context allows the given permission on the given object. For example, ‘SecurityCheckPermission("Add Documents, Images, and Files", this())’ would return true if the current user was authorized to create documents, images, and files in the current location.

SecurityGetUser() Return the current user object. This is normally the same as the ‘REQUEST.AUTHENTICATED_USER’ object. However, the ‘AUTHENTICATED_USER’ object is insecure since it can be replaced.

SecurityValidate([object], [parent], [name], [value]) Return true if the value is accessible to the current user. ‘object’ is the object the value was accessed in, ‘parent’ is the container of the value, and ‘name’ is the named used to access the value (for example, if it was obtained via ‘getattr’). You may omit some of the arguments, however it is best to provide all available arguments.

SecurityValidateValue(object) Return true if the object is accessible to the current user. This function is the same as calling ‘SecurityValidate(None, None, None, object)’.

str(object) Return a string containing a nicely printable representation of an object. For strings, this returns the string itself.

test(condition, result, [condition, result]... [default]) Takes one or more condition, result pairs and returns the result of the first true condition. Only one result is returned, even if more than one condition is true. If no condition is true and a default is given, the default is returned. If no condition is true and there is no default, None is returned.

unichr(number) Return a unicode string representing the value of number as a unicode character. This is the inverse of ord() for unicode characters.

unicode(string, [encoding], [errors]) Decodes string using the codec for encoding. Error handling is done according to errors. The default behavior is to decode UTF-8 in strict mode, meaning that encoding errors raise ValueError.

Attributes

None The ‘None’ object is equivalent to the Python built-in object ‘None’. This is usually used to represent a Null or false value.

See Also

- string module
- random module
- math module
- sequence module

7.25.4 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:

```xml
<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:

```xml
<dtml-if snake>
  The snake variable is true
</dtml-if>
```

Testing for expression conditions:

```xml
<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

7.25.5 in: Loops over sequences

The ‘in’ tag gives you powerful controls for looping over sequences and performing batch processing.

Syntax

‘in’ tag syntax:

```xml
<dtml-in SequenceVariable|expr="SequenceExpression”>
[<dtml-else>]
</dtml-in>
```

a commenting identifier at the end tag is allowed and will be ignored like:
same for ‘</dtml-if>’ and ‘</dtml-let>’

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:

```xml
<dtml-in objectValues>
   title: <dtml-var title><br>
</dtml-in>
```

Looping over two sets of objects, using prefixes:

```xml
<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:
Creating alternate colored table rows:

```xml
<table>
  <tr <dtml-if sequence-odd>bgcolor="#EEEEEE"
      <dtml-else>bgcolor="#FFFFFF"
      </dtml-if>>
    <td><dtml-var title></td>
  </tr>
</table>
```

Basic batch processing:

```xml
<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>
</p>
<p>
  <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.

### 7.25.6 let: Defines DTML variables

The ‘let’ tag defines variables in the DTML namespace.

**Syntax**

‘let’ tag syntax:

```xml
<dtml-let [Name=Variable][Name="Expression"]/>
```
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:**

```html
<dtml-let name="'Bob'" ids=objectIds>
  name: <dtml-var name>
  ids: <dtml-var ids>
</dtml-let>
```

**Using the ‘let’ tag with the ‘in’ tag:**

```html
<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

**7.25.7 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:

```html
<dtml-mime>
  [dtml-boundry]...
</dtml-mime>
```

The ‘mime’ tag is a block tag. The block is can be divided by one or more ‘boundry’ tags to create a multi-part MIME message. ‘mime’ tags may be nested. The ‘mime’ tag is most often used inside the ‘sendmail’ tag.
Attributes

Both the ‘mime’ and ‘boundry’ tags have the same attributes.

- **encode=string** MIME Content-Transfer-Encoding header, defaults to ‘base64’. Valid encoding options include ‘base64’, ‘quoted-printable’, ‘uuencode’, ‘x-uuencode’, ‘uue’, ‘x-uue’, and ‘7bit’. If the ‘encode’ attribute is set to ‘7bit’ no encoding is done on the block and the data is assumed to be in a valid MIME format.

- **type=string** MIME Content-Type header.

- **type_expr=string** MIME Content-Type header as a variable expression. You cannot use both ‘type’ and ‘type_expr’.

- **name=string** MIME Content-Type header name.

- **name_expr=string** MIME Content-Type header name as a variable expression. You cannot use both ‘name’ and ‘name_expr’.

- **disposition=string** MIME Content-Disposition header.

- **disposition_expr=string** MIME Content-Disposition header as a variable expression. You cannot use both ‘disposition’ and ‘disposition_expr’.

- **filename=string** MIME Content-Disposition header filename.

- **filename_expr=string** MIME Content-Disposition header filename as a variable expression. You cannot use both ‘filename’ and ‘filename_expr’.

- **skip_expr=string** A variable expression that if true, skips the block. You can use this attribute to selectively include MIME blocks.

Examples

Sending a file attachment:

```
<dtm1-sendmail>
 To: <dtm1-var recipient>
 Subject: Resume
<dtm1-mime type="text/plain" encode="7bit">
 Hi, please take a look at my resume.
<dtm1-boundary type="application/octet-stream" disposition="attachment"
 encode="base64" filename_expr="resume_file.getId()"><dtm1-var expr="resume_file.read()"
 ..."/></dtm1-mime>
</dtm1-sendmail>
```

See Also

- Python Library mimetools

7.25.8 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

7.25.9 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}>```
7.25.10 sendmail: Sends email with SMTP

The ‘sendmail’ tag sends an email message using SMTP.

Syntax

‘sendmail’ tag syntax:

```html
<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 an 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:

```html
<dtml-sendmail mailhost="mailhost">
To: <dtml-var recipient>
From: <dtml-var sender>
Subject: <dtml-var subject>

Dear <dtml-var recipient>,

Your 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
7.25.11 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:

```xml
<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:

```python
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>
    </dtml-sqlgroup>
```

If ‘first’ is ‘Bob’ and ‘last’ is ‘Smith, McDonald’ it renders:

```sql
select * from employees
where
    (first = 'Bob'
    and
    last in ('Smith', 'McDonald'))
```

If ‘salary’ is 50000 and ‘last’ is ‘Smith’ it renders:

```sql
select * from employees
where
    (salary > 50000.0
    and
    last = 'Smith')
```

(continues on next page)
Nested ‘sqlgroup’ tags:

```dtml-sqlgroup
select * from employees
<dtml-sqlgroup where>
  <dtml-sqltest first op="like" type="nb">
    <dtml-sqltest last op="like" type="nb">
      <dtml-sqltest salary op="gt" type="float">
    </dtml-sqltest>
  </dtml-sqltest>
</dtml-sqlgroup>
```

Given sample arguments, this template renders to SQL like so:

```dtml-sqltest
select * from employees
where
  (name like 'A*' and last like 'Smith')
  or salary > 20000.0
```

See Also

- sqltest tag

### 7.25.12 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
<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:

```sql
select * from employees
where <dtml-sqltest name type="nb">
```

If the ‘name’ variable is ‘Bob’ then this renders:

```sql
select * from employees
where name = 'Bob'
```

Multiple values:

```sql
select * from employees
where <dtml-sqltest empid type=int multiple>
```

If the ‘empid’ variable is ‘(12,14,17)’ then this renders:

```sql
select * from employees
where empid in (12, 14, 17)
```

See Also

- sqlgroup tag
- sqlvar tag
7.25.13 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

7.25.14 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:

```xml
<dtml-tree>
  <dtml-var title_or_id>
</dtml-tree>
```

Display a tree rooted in another object, using a custom branches method:

```xml
<dtml-tree expr="folder.object" branches="objectValues">
  Node id : <dtml-var getId>
</dtml-tree>
```

7.25.15 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:

  ```xml
  <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:

  ```xml
  <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:

```html
<dtm-try>
<dtm-var expr="1/0">
<dtm-except ZeroDivisionError>
You tried to divide by zero.
</dtm-except>
</dtm-try>
```

Returning information about the handled exception:

```html
<dtm-try>
<dtm-call dangerousMethod>
<dtm-except>
An error occurred.
Error type: <dtm-var error_type>
Error value: <dtm-var error_value>
</dtm-except>
</dtm-try>
```

Using finally to make sure to perform clean up regardless of whether an error is raised or not:

```html
<dtm-call acquireLock>
<dtm-try>
<dtm-call someMethod>
<dtm-finally>
<dtm-call releaseLock>
</dtm-try>
</dtm-call>
```

See Also

- raise tag
- Python Tutorial Errors and Exceptions
- Python Built-in Exceptions

7.25.16 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-call dangerousOperation>
</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

7.25.17 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.

  **collection-length** The length of the variable, assuming it is a sequence.

**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:

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

Truncation:

```html
<dtml-var colors size=10 etc="", etc."> red yellow, etc.
```

will produce the following output if colors is the string ‘red yellow green’:

```html
red yellow, etc.
```

C-style string formatting:

```html
<dtml-var expr="23432.2323" fmt="%.2f"> 386 Chapter 7. The Zope Book
```
Inserting a variable, *link*, inside an HTML ‘A’ tag with the entity syntax:

```html
<a href="&dtml-link;">Link</a>
```

Inserting a link to a document ‘doc’, using entity syntax with attributes:

```html
<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.

### 7.25.18 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:

```html
<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:

```html
<dtml-with REQUEST only>
  <dtml-if id>
    <dtml-var id>
  </dtml-if>
  <dtml-else>
    'id' was not in the request.
  </dtml-else>
</dtml-with>
```

Pushing the first child on the DTML namespace:

```html
<dtml-with expr="objectValues()[0]">  First child’s id: <dtml-var id>
</dtml-with>
```
See Also

- let tag

7.26 Appendix B: API Reference

Attention: This document was written for Zope 2.

7.26.1 Introduction

This reference describes the interfaces to the most common set of basic Zope objects. This reference is useful while writing Page Templates, DTML, Python scripts, and Product code.

The intended audience is able to read simple Python code and has at least passing experience with object-oriented programming.

The reference is not a tutorial. Nor is it a substitute for reading the rest of the Zope Book. Examples, where they are provided, are intended to be illustrative, but not comprehensive.

7.26.2 Sorry

The manually maintained API reference wasn’t such a good idea.

Converting it from the original source of structured text to reStructuredText was too much work to be done. We will look into auto-generating the API documentation from docstrings at some point.

Reading the code is your best bet for now.

7.27 Appendix C: Zope Page Templates Reference

Attention: This document was written for Zope 2.

Zope Page Templates are an HTML/XML generation tool. This appendix is a reference to Zope Page Templates standards: Template 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.

7.27.1 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:
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:

```xml
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`
Since the *on-error* statement is only invoked when an error occurs, it does not appear in the list.

It may not be apparent that there needs to be an ordering. The reason that there must be one is that TAL is XML based. The XML specification specifically states that XML processors are free to rewrite the terms. In particular, you cannot assume that attributes of an XML statement will be processed in the order written, particularly if there is another preprocessor involved. To avoid needless proliferation of tags, and still permit unambiguous execution of complex TAL, a precedence order was chosen according to the following rationale.

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.

### 7.27.2 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:

```html
<a href="/sample/link.html"
thal:attributes="href context/sub/absolute_url">
```

Replacing two attributes:

```html
<textarea
  rows="80" cols="20"
thal:attributes="rows request/rows;cols request/cols">
```

### 7.27.3 condition: Conditionally insert or remove an element

**Syntax**

```plaintext
tal:condition syntax:
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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):

```html
<p tal:condition="request/message | nothing"
thal:content="request/message">message goes here</p>
<p tal:condition="exists:request/message"
thal:content="request/message">message goes here</p>
```

Test for alternate conditions:

```html
<div tal:repeat="item python:range(10)">
  <p tal:condition="repeat/item/even">Even</p>
  <p tal:condition="repeat/item/odd">Odd</p>
</div>
```
7.27.4 content: Replace the content of an element

Syntax

tal:content syntax:

\[
\text{argument} ::= (\['\text{text}'\] | 'structure') \text{expression}
\]

Description

Rather than replacing an entire element, you can insert text or structure in place of its children with the \textit{tal:content} statement. The statement argument is exactly like that of \textit{tal:replace}, and is interpreted in the same fashion. If the expression evaluates to \textit{nothing}, the statement element is left childless. If the expression evaluates to \textit{default}, then the element’s contents are unchanged.

The default replacement behavior is \textit{text}, which replaces angle-brackets and ampersands with their HTML entity equivalents. The \textit{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 (e.g., text submitted via a web form), which is the reason that it is not the default.

Examples

Inserting the user name:

\[
<p \texttt{tal:content="user/getUserName">Fred Farkas</p>
\]

Inserting HTML/XML:

\[
<p \texttt{tal:content="structure context/getStory">marked <b>up</b> content goes here.</p>
\]

7.27.5 define: Define variables

Syntax

tal:define syntax:

\[
\text{argument} ::= \text{define\_scope} [';' \text{define\_scope}]*
\text{define\_scope} ::= (\['\text{local}'\] | 'global') \text{define\_var}
\text{define\_var} ::= \text{variable\_name} \text{expression}
\text{variable\_name} ::= \text{Name}
\]

Note: If you want to include a semi-colon (;) in an ‘expression’, it must be escaped by doubling it (;;).

Description

The \textit{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)"
```

### 7.27.6 omit-tag: Remove an element, leaving its contents

**Syntax**

`tal:omit-tag` syntax:

```
arument ::= [ 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:
7.27.7 on-error: Handle errors

Syntax

tal:on-error syntax:

\[\text{argument} ::= ([\text{'text'}] | \text{'structure'}) \text{ expression}\]

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:

```xml
<b tal:on-error="string: Username is not defined!"
    tal:content="context/getUsername">Ishmael</b>
```

Removing elements with errors:

```xml
<b tal:on-error="nothing"
    tal:content="context/getUsername">Ishmael</b>
```

Calling an error-handling script:

```xml
<div tal:on-error="structure context/errorScript">
  ...
</div>
```

Here's what the error-handling script might look like:
7.27.8 repeat: Repeat an element

Syntax

tal:repeat syntax:

<table>
<thead>
<tr>
<th>argument</th>
<th>variable_name expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>variable_name</td>
<td>Name</td>
</tr>
</tbody>
</table>

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**- 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 groups 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
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:

```html
<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:

```html
<table>
  <tr tal:repeat="item context/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:

```html
<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. Separate groups of objects by meta-type by drawing a rule between them:
Note, the objects in the above example should already be sorted by meta-type.

### 7.27.9 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 `&`, `<` 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>
```

### 7.27.10 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 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.
- **context** - 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`, `context`, `container`, `template`, `request`, `user`, and `modules` are optional names supported by Zope, but are not required by the TALES standard.

### 7.27.11 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.

### 7.27.12 TALES Nocall expressions

**Syntax**

Nocall expression syntax:

```
ocall_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:

```html
<span tal:define="doc nocall:context/aDoc"
   tal:content="string:${doc/getId}: ${doc/title}"
   Id: Title
</span>
```

Using nocall expressions on a functions:

```html
<p tal:define="join nocall:modules/string/join">
This example defines a variable:: join which is bound to the string.join function.
```

7.27.13 TALES Not expressions

Syntax

Not expression syntax:

```
not_expression ::= 'not:' expression
```

Description

Not expression evaluates 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:

```html
<p tal:condition="not:context/objectIds">
There are no contained objects.
</p>
```
7.27.14 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
context/some-file 2009_02.html.tar.gz/foo
root/to/branch | default
request/name | string:Anonymous Coward
context/?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 sub-object named by the path segment. Sub-objects 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 TALES 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 TALES 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:

```html
<span tal:replace="request/cookies/pref | context/pref">preference</span>
```

Inserting the user name:

```html
<p tal:content="user/getUserName">User name</p>
```

### 7.27.15 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.

**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 to avoid accidental long execution times.

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):

```python
<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):

```python
<p tal:content="python:user.getUserName().capitalize()">User Name</p>
```

Basic math (convert an image size to megabytes):

```python
<p tal:content="python:image.getSize() / 1048576.0">12.2323</p>
```

String formatting (format a float to two decimal places):

```python
<p tal:content="python:'%0.2f' % size">13.56</p>
```

**7.27.16 TALES String expressions**

**Syntax**

String expression syntax:
**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:

```xml
<span tal:replace="string:$this and $that">
    Spam and Eggs
</span>
```

Using paths:

```xml
<p tal:content="string:total: ${request/form/total}">
    total: 12
</p>
```

Including a dollar sign:

```xml
<p tal:content="string:cost: $$cost">
    cost: $42.00
</p>
```

**7.27.17 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:

```xml
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.

### 7.27.18 define-macro: Define a macro

#### Syntax

define-macro syntax:

```plaintext
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:

```plaintext
master.html/macros/header
```

#### Examples

Simple macro definition:

```html
<p metal:define-macro="copyright">
  Copyright 2009, <em>Foobar</em> Inc.
</p>
```

### 7.27.19 define-slot: Define a macro customization point

#### Syntax

define-slot syntax:

```plaintext
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:

```xml
<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.

7.27.20 fill-slot: Customize a macro

Syntax

`metal:fill-slot` syntax:

```plaintext
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:

```xml
<p metal:define-macro="hello">
    Hello <b metal:define-slot="name">World</b>
</p>
```

You can fill the `name` slot like so:

```xml
<p metal:use-macro="container/master.html/macros/hello">
    Hello <b metal:fill-slot="name">Kevin Bacon</b>
</p>
```
7.27.21 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.

7.27.22 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 `<span>` 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 \(<\text{tag}/>\) form.

Certain boolean attributes, such as \texttt{checked} and \texttt{selected}, are treated differently by \texttt{tal:attributes}. The value is treated as true or false (as defined by \texttt{tal:condition}). The attribute is set to \texttt{attr="attr"} in the true case and omitted otherwise. If the value is \texttt{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:

\begin{verbatim}
<input type="checkbox" checked tal:attributes="checked default" />
<input type="checkbox" tal:attributes="checked string:yes" />
<input type="checkbox" tal:attributes="checked python:42" />
\end{verbatim}

will render as:

\begin{verbatim}
<input type="checkbox" checked="checked" />
\end{verbatim}

while each of these:

\begin{verbatim}
<input type="checkbox" tal:attributes="checked default" />
<input type="checkbox" tal:attributes="checked string:" />
<input type="checkbox" tal:attributes="checked nothing" />
\end{verbatim}

will render as:

\begin{verbatim}
<input type="checkbox" />
\end{verbatim}

This works correctly in all browsers in which it has been tested.

7.28 Appendix D: Zope Resources

\textbf{Attention: } This document was written for Zope 2.

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.

7.28.1 Zope Web Sites

\textbf{Zope.org} is the official Zope website. It has downloads, documentation, news, and lots of community resources.

\textbf{DZUG} was started as the main community site for the German Zope community and combines documentation translated to German, downloads, a portal for the various regional German Zope User Groups as well as information about Zope-related events in Europe.

\textbf{Zope Italia} forms the focal point for the Italian Zope community with news, events information and local Zope group contacts.

7.28.2 Zope Documentation

\textbf{ZopeWiki - A wiki for the Zope community} is a community-run Zope documentation website set up by Simon Michael, author of the famus ZWiki wiki product for Zope.
Zope Developer’s Guide teaches you how to write Zope products. It is somewhat outdated but contains some nuggets you don’t find elsewhere.

### 7.28.3 (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.
- **Zope: Timely, Practical, Reliable** written by Pierre Julien Grizel.

**The Zope Book** is the hardcover version of the original edition Zope Book on which this text is based.

### 7.28.4 Mailing Lists

mail.zope.org maintains a collection of the many Zope mailing lists.

### 7.28.5 Python Information

Python.org has lots of information about Python including a tutorial and reference documentation.

### 7.29 Appendix E: DTML Name Lookup Rules

**Attention:** This document was written for Zope 2.

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:

```python
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 (i.e. **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.

### 7.30 Contributions

**Attention:** This document was written for Zope 2.

Contributors to this book include Amos Latteier, Michel Pelletier, Chris McDonough, Evan Simpson, Tom Deprez, Paul Everitt, Bakhtiar A. Hamid, Geir Baekholt, Thomas Reulbach, Paul Winkler, Peter Sabaini, Andrew Veitch, Kevin Carlson, Joel Burton, John DeStefano, Tres Seaver, Hanno Schlichting, 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” and the “Searching and Categorizing Content” 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.

Joel Burton rewrote the ZCatalog chapter late in the 2.6 book’s lifetime.

Dario Lopez-Kästen updated the “Introducing Zope” chapter for the 2.7 edition.

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.

Jo <jo at winfix dot it> has contributed a number of spelling corrections.

John DeStefano edited chapters of the book in a post-2.7-edition mode.

Tres Seaver moved the text into the Zope Subversion repository, and helped with the conversion of the text from Structured Text to ReStructured Text.
Hanno Schlichting did the remainder of the ReStructured Text conversion, completed the integration with Sphinx and rewrote many chapters for Zope 2.12.

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

8.1.1 Overview

Zope is a free and open-source, object-oriented web application server written in the Python programming language. The term ZOPE is an acronym for “Z Object Publishing Environment” (the Z doesn’t really mean anything in particular). However, nowadays ZOPE is simply written as Zope. It has three distinct audiences.

Site Managers  Individuals who use of Zope’s “out of the box” features to build websites. This audience is interested in making use of Zope’s existing array of features to create content management solutions. They are generally less concerned about code reuse than the speed with which they can create a custom application or website.

Developers  Individuals who wish to extend Zope to create highly customized solutions. This audience is likely interested in creating highly reusable custom code that makes Zope do something new and interesting.

Administrators  Individuals responsible for keeping a Zope site running and performing installations and upgrades.

This guide is intended to document Zope for the second audience, Developers, as defined above. If you fit more into the “user” audience defined above, you’ll probably want to start by reading The Zope Book .

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

8.1.2 Organization of the book

This book describes Zope’s services to the developer from a hands on, example-oriented standpoint. This book is not a complete reference to the Zope API, but rather a practical guide to applying Zope’s services to develop and deploy your own web applications. This book covers the following topics:

Getting Started  This chapter provides a brief overview of installation and getting started with application development.
Components and Interfaces  Zope uses a component-centric development model. This chapter describes the component model in Zope and how Zope components are described through interfaces.

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

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

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

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

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

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

8.2 Getting Started

8.2.1 Introduction

This chapter covers the installation of Zope and getting started with the development of a simple application. This guide uses a build system called Buildout to build the application.

8.2.2 Prerequisites

Make sure you have Python installed. Version 3.6 or higher is recommended.

Creating and activating a VirtualEnv is recommended.

In order to use buildout, you have to install the zc.buildout package.

$ pip install zc.buildout

8.2.3 Directory structure

To begin application development, create a directory structure for the Python packages and build related files.

$ mkdir poll
$ mkdir poll/poll_build
$ mkdir poll/poll.main

All build related files will be added inside the poll_build directory, whereas the main Python package goes into the poll.main directory.

8.2.4 Installing Zope using zc.buildout

Zope is distributed in egg format. To install Zope and create an instance, create a buildout configuration file (poll/poll_build/buildout.cfg) with following content.
The [zope4] part uses zc.recipe.egg which will download Zope and all its dependencies. It will create few console scripts inside the bin directory.

After updating the buildout configuration, you can run the buildout command to build the system.

```bash
$ buildout
```

The initial build will take some time to complete.

### 8.2.5 Creating the instance

Once the build is complete, you can create an instance as follows.

```bash
$ bin/mkwsgiinstance -d .
```

### 8.2.6 Running the instance

Once you got a Zope instance, you can run it like this.

```bash
$ bin/runwsgi etc/zope.ini
```

Now, Zope is running. You can convince yourself by visiting the following URL.

```text
http://localhost:8080
```

You can also visit the administration area.

Use the user name and password you set earlier.

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

When you have a look at the drop-down box in the top right corner, you see a list of objects you may create.

In the next section we will create the poll application. Later, we will make it installable, too.

### 8.2.7 Developing the main package

Now, we can move to the poll.main directory to create the main package to develop the application. We will develop the entire application inside the poll.main package. For bigger projects, it is recommended to split packages logically and maintain the dependencies between the packages properly.

```bash
$ cd ../poll.main
```
In order to create an egg distribution, we need to create a `setup.py` and a basic directory structure. We are going to place the Python package inside the `src` directory.

```
$ touch setup.py
$ mkdir src
$ mkdir src/poll
$ mkdir src/poll/main
$ touch src/poll/__init__.py
$ touch src/poll/main/__init__.py
$ touch src/poll/main/configure.zcml
```

The last file is a configuration file. The `.zcml` file extension stands for Zope Configuration Markup Language.

To declare `poll` as a namespace package, we need to add following code to `src/poll/__init__.py`.

```
__import__('pkg_resources').declare_namespace(__name__)
```

Next, we need to add the minimum metadata required for the package in `setup.py`.

```
from setuptools import setup, find_packages

setup(  
    name="poll.main",  
    version="0.1",  
    packages=find_packages("src"),  
    package_dir="": "src"),  
    namespace_packages=["poll"],  
    install_requires=["setuptools",  
                        "Zope"],
)
```

We need to edit two more files to be recognized by Zope. First, define the `initialize` callback function in `src/poll/main/__init__.py`.

```
def initialize(registrar):
    pass
```

And, in the ZCML file (`src/poll/main/configure.zcml`), add these few lines.

```
<configure xmlns="http://namespaces.zope.org/five">
  <registerPackage package="." initialize=".initialize" />
</configure>
```

### 8.2.8 Creating an installable application

We need three things to make an installable application.

- A form object created as Zope Page Template (`manage_addPollMain`)
- A function to define the form action (`addPollMain`)
- A class to define the toplevel application object (`PollMain`).

Finally, we need to register the class along with the form and add the function using the `registrar` object passed to the `initialize` function.
We can define all these things in `app.py` and the form template as `manage_addPollMain_form.zpt`.

```bash
$ touch src/poll/main/app.py
$ touch src/poll/main/manage_addPollMain_form.zpt
```

Here is the code for `app.py`...

```python
from OFS.Folder import Folder
from Products.PageTemplates.PageTemplateFile import PageTemplateFile

class PollMain(Folder):
    meta_type = "POLL"

manage_addPollMain = PageTemplateFile("manage_addPollMain_form", globals())

def addPollMain(context, id):
    """
    context._setObject(id, PollMain(id))
    return "POLL Installed: %s" % id
```

... and for `manage_addPollMain_form.zpt`:

```html
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:tal="http://xml.zope.org/namespaces/tal">
<body>
    <h2>Add POLL</h2>
    <form action="addPollMain" method="post">
        Id: <input type="text" name="id" /><br />
        Title: <input type="text" name="title" /><br />
        <input type="submit" value="Add" />
    </form>
</body>
</html>
```

Finally, we can register it within `src/poll/main/__init__.py`:

```python
from poll.main.app import PollMain, manage_addPollMain, addPollMain

def initialize(registrar):
    registrar.registerClass(
        PollMain,
        constructors=(manage_addPollMain, addPollMain)
    )
```

The application is now ready to install. But we need to make some changes in `poll_build`, so it gets installed along Zope.

### 8.2.9 Updating the build config

First, in the `[buildout]` section of `buildout.cfg` we need to mention that `poll.main` is locally developed. Otherwise, buildout will try to get the package from package index server, by default that is https://pypi.org/.

---

**8.2. Getting Started**
Also, we need to add `poll.main` to the `eggs` option in the `[zope4]` section.

The final `buildout.cfg` will look like this.

```ini
[buildout]
develop = ../poll.main
extends = https://zopefoundation.github.io/Zope/releases/master/versions-prod.cfg
parts =
    zope4

[zope4]
recipe = zc.recipe.egg
eggs =
    Zope
    poll.main
```

To make these changes effective, run the buildout again.

```bash
$ buildout
```

Finally, we have to include our package within `poll_build/etc/site.zcml`.

```xml
...<include package="poll.main" />
...```

Now, we can run the application instance again.

```bash
$ bin/runwsgi etc/zope.ini
```

### 8.2.10 Adding an application instance

Visit the ZMI (http://localhost:8080/manage) and select POLL from the drop-down box. It will display the add-form created earlier. Enter `poll` in the ID field and submit the form. After submitting, it should display a message: “POLL Installed: poll”.

### 8.2.11 Adding and index page for the POLL application

In this section we will add a main page to the POLL application, so that we can access the POLL application like this: http://localhost:8080/poll.

First, create a file named `index_html.zpt` inside `poll.main/src/poll/main` with content like this:
Now add an attribute named `index_html` inside `PollMain` class like this:

```python
class PollMain(Folder):
    meta_type = "POLL"
    index_html = PageTemplateFile("index_html", globals())
```

After restarting Zope, you can see that it displays the main page when you access: `http://localhost:8080/poll`.

### 8.2.12 Summary

This chapter covered the installation of Zope and the beginning of the development of a simple project in Zope.

### 8.3 Components and Interfaces

Zope uses a component architecture internally in many places. Zope components are nothing but Python objects with interfaces that describe them. As a Zope developer you can use interfaces right now to build your Zope components.

#### 8.3.1 Zope Components

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

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

```python
from zope.interface import Interface
from zope.interface import implementer

class IHello(Interface):
    """The Hello interface provides greetings."""

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

@implementer(IHello)
class HelloComponent:
    def hello(self, name):
        return "hello %s!" % name
```
Let’s take a look at this step by step. Here, you see two Python class statements. The first class statement creates the interface, and the second class statement creates the implementation.

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

The second class statement creates the HelloComponent class. This class is the actual component that does what IHello describes. This is usually referred to as the implementation of IHello. In order for you to know what interfaces HelloComponent implements, it must somehow associate itself with an interface. The implemeneter decorator above the class does just that. It says, “I implement these interfaces”. In this case, HelloComponent asserts that it implements one interface, IHello.

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

```python
import xmlrpclib

@implementer(IHello)
class XMLRPCHello:
    def hello(self, name):
        """Delegates the hello call to a remote object using XML-RPC."
        s = xmlrpclib.Server('your/rpc/server')
        return s.hello(name)
```

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

And that’s all there is to components, really. The rest of this chapter describes interfaces and how you can work with them from the perspective of components. In Chapter 5, we’ll put all this together into a Zope product.

### 8.3.2 Python Interfaces

An interface describes the behavior of an object by containing useful information about the object. This information includes:

- Prose documentation about the object. In Python terms, this is called the “doc string” of the interface. In this element, you describe how the object works in prose language and any other useful information about the object.

- Descriptions of attributes. Attribute descriptions include the name of the attribute and prose documentation describing the attributes usage.

- Descriptions of methods. Method descriptions can include:
  - Prose “doc string” documentation about the method and its usage.
  - A sequence of parameter objects that describes the parameters expected by the method.

- Optional tagged data. Interface objects (and their attributes, methods, and method parameters) can have optional, application specific tagged data associated with them. Examples uses for this are security assertions, pre/post conditions, unit tests, and other possible information you may want to associate with an Interface or its attributes.

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

### 8.3.3 Why Use Interfaces?

Interfaces solve a number of problems that arise while developing large systems with lots of developers.
• Developers waste a lot of time looking at the source code of your system to figure out how objects work. This is even worse if someone else has already wasted their time doing the same thing.

• Developers who are new to your system may misunderstand how an object works, causing, and possibly propagating, usage errors.

• Because an object’s interface is inferred from the source, developers may end up using methods and attributes that are meant for “internal use only”.

• Code inspection can be hard, and very discouraging to novice programmers trying to understand code written by gurus.

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

### 8.3.4 Creating Interfaces

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

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

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

```python
from zope.interface import Interface

class IHello(Interface):
    def hello(name):
        """Say hello to the world""
```

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

```python
>>> IHello
<InterfaceClass __main__.IHello>
```

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

```python
from zope.interface import implementer

@implementer(IHello)
class HelloComponent:
    def hello(self, name):
        return "Hello %s!" % name
```

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

```python
@implementer(IHello, IItem)
class HelloComponent:
    ...
```
8.3.5 The Interface Model

Interfaces can extend other interfaces. For example, let’s extend the IHello interface by adding an additional method:

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

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

ISmartHello extends the IHello interface. It does this by using the same syntax a class would use to subclass another class.

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

```python
>>> ISmartHello.getBases()
(<InterfaceClass __main__.IHello>,)
```

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

```python
>>> ISmartHello.extends(IHello)
True
>>> class ISandwich(Interface):
    ...
pass
>>> ISmartHello.extends(ISandwich)
False
```

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

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

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

8.3.6 Querying an Interface

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

```python
>>> IHello.names()
dict_keys(['hello'])
```

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

For example:

```python
>>> IHello.namesAndDescriptions()
dict_items([('hello', <zope.interface.interface.Method object at 0x7fc6875110f0>)]
```
Note: You cannot access the Method object by index, as namesAndDescriptions returns a dict_view.
You can either use list or next and iter on the result.

As you can see, the “description” of the Interface’s item is a Method object. Description objects can be either Attribute or Method objects. Attribute, Method and Interface objects implement the following interface:

- `getName()` -- Returns the name of the object.
- `getDoc()` -- Returns the documentation for the object.

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

- `getSignatureInfo()` -- Returns a dictionary describing the method parameters.
- `getSignatureString()` -- Returns a human-readable string representation of the method’s signature.

For example:

```python
>>> m = list(IHello.namesAndDescriptions())[0][1]
>>> m
<zope.interface.interface.Method object at 0x7fc6875110f0>
>>> m.getSignatureString()
'(name)'
>>> m.getSignatureInfo()
{'positional': ('name',), 'required': ('name',), 'optional': {},
 'varargs': None, 'kwargs': None}
```

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

8.3.7 Checking Implementation

You can ask an interface if a certain class that you hand it implements that interface. For example, say you want to know if the HelloComponent class implements IHello:

```python
IHello.implementedBy(HelloComponent)
```

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

```python
IHello.providedBy(my_hello_instance)
```

This would also return true if my_hello_instance was an instance of HelloComponent, or any other object of a class that implemented the IHello interface.

8.3.8 Conclusion

Interfaces provide a simple way to describe your Python objects. By using interfaces you document capabilities of objects.
8.4 Object Publishing

8.4.1 Introduction

Zope puts your objects on the web. This is called object publishing. One of Zope’s unique characteristics is the way it allows you to walk up to your objects and call methods on them with simple URLs. In addition to HTTP, Zope makes your objects available via XML-RPC.

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

8.4.2 HTTP Publishing

Zope 4 no longer ships with a built-in web server, so when you want to interact with Zope via browser you have to setup a WSGI server.

Note: For usage on a production server you will probably want to setup a reverse proxy in front of the WSGI server.

The WSGI server receives the request and hands it over to Zope, where it is processed by ZPublisher, which is Zope’s object publisher. ZPublisher is a kind of light-weight ORB (Object Request Broker). It takes the request and locates an object to handle the request. The publisher uses the request URL as a map to locate the published object. Finding an object to handle the request is called traversal, since the publisher moves from object to object as it looks for the right one. Once the published object is found, the publisher calls a method on the published object, passing it parameters as necessary. The publisher uses information in the request to determine which method to call and what parameters to pass. The process of extracting parameters from the request is called argument marshalling. The published object then returns a response, which is passed back to the WSGI server. Finally, the WSGI server passes the response back to your web browser.

The publishing process is summarized in [2-1]

![Diagram of object publishing process](image)

Fig. 1: 2.1 Object publishing
Typically the published object is a persistent object that the published module loads from the ZODB. See Chapter 6 for more information on the ZODB.

This chapter will cover all the steps of object publishing in detail.

To summarize, object publishing consists of the main steps:

1. A request is sent to the publisher.
2. The publisher locates the published object using the request URL as a map.
3. The publisher calls the published object with arguments from the request.
4. The publisher interprets the results and passes them back.

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

**URL Traversal**

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

```python
class Classification:
    ...

class Animal:
    ...

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

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

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

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

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

Now let’s take a closer look at traversal.

**Publishable Object Requirements**

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

Another requirement is that a publishable object must not have a name that begins with an underscore. These two restrictions are designed to keep private objects from being published.
Fig. 2: 2.2 Traversal path through an object hierarchy
Finally, published objects cannot be Python modules.

**Traversal Methods**

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

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

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

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

```python
def __bobo_traverse__(self, request, key):
    """Return subobjects depending on cookie contents.""
    if request.cookies.has_key('special'):
        return self.special_subobjects.get(key, None)
    return self.normal_subobjects.get(key, None)
```

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

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

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

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

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

**Publishing Methods**

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

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

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

- Stringifying the published object – If the published object isn’t callable, and doesn’t have a default method, the publisher publishes it using the Python `str` function to turn it into a string.
After the response method has been determined and called, the publisher must interpret the results.

**Character Encodings for Responses**

If the published method returns an object of type *binary*, the publisher will use it directly as the body of the response. Things are different if the published method returns a unicode string, because the publisher has to apply some character encoding. The published method can choose which character encoding it uses by setting a `Content-Type` response header which includes a `charset` property (setting response headers is explained later in this chapter). A common choice of character encoding is UTF-8, which is also the default encoding.

If the `Content-Type` header does not include a charset or is not set at all, the default encoding is set.

If you want to manually set a `Content-Type` header you have to set a value like `text/html; charset=UTF-8`.

**HTTP Responses**

Usually, the published method returns a string which is considered the body of the HTTP response. The response headers can be controlled by calling methods on the response object, which is described later in the chapter.

---

**Note:** When the return value is empty, e.g. an empty list, instead of returning an empty page, Zope issues a header with a 204 status code.

Depending on the used client, it looks like nothing happens.

Optionally, the published method can return a tuple with the title and the body of the response. In this case, the publisher returns a generated HTML page, with the first item of the tuple used for the value of the HTML `title` tag of the page, and the second item as the content of the HTML `body` tag.

For example a response of:

```
("my_title", "my_text")
```

is turned into this HTML page:

```
<html>
<head><title>my_title</title></head>
<body>my_text</body>
</html>
```

**Controlling Base HREF**

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

Consider this example:

```
class Example:
    """example class""

    def one(self):
        """render page one""
        return """"<html>
            <head><title>one</title></head>
            <body>
```

(continues on next page)
However, the default method `index_html` presents a problem. Since you can access `index_html` without specifying the method name in the URL, relative links returned by `index_html` won’t work right.

For example:

```python
class Example:
    """example class"""
    def index_html(self):
        """render default view"""
        return """<html>
            <head><title>one</title></head>
            <body>
                <a href="one">one</a><br>
                <a href="two">two</a>
            </body>
        </html>"
...
```

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

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

**Response Headers**

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

**Pre-Traversal Hook**

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

If your object has a method named `__before_publishing_traverse__`, the publisher will call it with the current object and the request before traversing your object. Most often your method will change the request. The publisher ignores anything you return from the pre-traversal hook method.
The `ZPublisher.BeforeTraverse` module contains some functions that help you register pre-traversal callbacks. This allows you to perform fairly complex callbacks to multiple objects when a given object is about to be traversed.

**Traversal and Acquisition**

**Note:** Simply put, acquisition means that a Zope object can acquire any attribute of its parents.

For detailed information about acquisition please refer to chapter 7.

Acquisition affects traversal in several ways. The most obvious way is in locating the next object in a path. As we discussed earlier, the next object during traversal is often found using `getattr`. Since acquisition affects `getattr`, it will affect traversal. The upshot is that when you are traversing objects that support implicit acquisition, you can use traversal to walk over acquired objects.

Consider the following object hierarchy:

```python
from Acquisition import Implicit

class Node(Implicit):
    ...

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

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

Mixing acquisition and traversal can get complex. In general you should limit yourself to constructing URLs which use acquisition to acquire along containment, rather than context lines.

It’s reasonable to publish an object or method that you acquire from your container, but it’s probably a bad idea to publish an object or method that your acquire from outside your container.

For example:

```python
from Acquisition import Implicit

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

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

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

(continues on next page)
basket=Basket()
basket.apple=Fruit()
basket.carrot=Vegetable()

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

A second usage of acquisition in traversal concerns the request. The publisher tries to make the request available to the published object via acquisition. It does this by wrapping the first object in an acquisition wrapper that allows it to acquire the request with the name ‘REQUEST’.

This means that you can normally acquire the request in the published object like so:

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

or like so:

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

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

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

**Traversal and Security**

As the publisher moves from object to object during traversal it makes security checks. The current user must be authorized to access each object along the traversal path. The publisher controls access in a number of ways. For more information about Zope security, see chapter 8 “Security”.

**Basic Publisher Security**

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

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

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

Some objects such as functions and methods historically did not support creating attributes. Consequently, if the object has no __roles__ attribute, the publisher will look for an attribute on the object’s parent with the name of the object followed by __roles__. For example, a function named getInfo would store its roles in its parent’s getInfo__roles__ attribute.

If an object has a __roles__ attribute that is not empty and not None, the publisher tries to find a user database to authenticate the user. It searches for user databases by looking for an __allow_groups__ attribute, first in the published object, then in the previously traversed object, and so on until a user database is found.
When a user database is found, the publisher attempts to validate the user against the user database. If validation fails, then the publisher will continue searching for user databases until the user can be validated or until no more user databases can be found.

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

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

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

The validate method returns a user object if succeeds, and `None` if it cannot validate the user. See Chapter 8 for more information on user objects. Normally, if the validate method returns `None`, the publisher will try to use other user databases, however, a user database can prevent this by raising an exception.

If validation fails, Zope will return an HTTP header that causes your browser to display a user name and password dialog.

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

### Zope Security

The publisher uses acquisition to locate user folders and perform security checks. The upshot of this is that your published objects must inherit from `Acquisition.Implicit` or `Acquisition.Explicit`.

**Note:** For more information on `Acquisition`, visit one of the following resources:
- chapter 7 “Acquisition” of this Zope Developer’s Guide
- chapter 8 “Acquisition” of *The Zope Book*
- the excellent readme of the “Acquisition” package

Also, when traversing, each object must be returned in an acquisition context. This is done automatically when traversing via `getattr`, but you must wrap traversed objects manually when using `__getitem__` and `__bobo_traverse__`.

For example:

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

def __bobo_traverse__(self, name, request):
    ...
    next_object=self._get_next_object(name)
    return next_object.__of__(self)
```

Finally, traversal security can be circumvented with the `__allow_access_to_unprotected_subobjects__` attribute as described in Chapter 8, “Security”.

### Calling the Published Object

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

```python
def greet(name):
    """Greet someone by name."""
    return "Hello, %s!" % name
```

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

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

**Marshalling Arguments from the Request**

Zope responds to requests, specified via URL, request headers and an optional request body. A URL consists of various parts, among them a `path` and a `query`, see RFC 2396 for details.

Zope uses the `path` to locate an object, method or view for producing the response (this process is called *traversal*) and `query` - if present - as a specification for request parameters. Additionally, request parameters can come from the optional request body.

Zope preprocesses the incoming request information and makes the result available in the so called `request` object. This way, the response generation code can access all relevant request information in an easy and natural (pythonic) way. Preprocessing transforms the request parameters into request (or form) variables. They are made available via the request object’s `form` attribute (a dict) or directly via the request object itself, as long as they are not hidden by other request information.

The request parameters coming from the *query* have the form `name=value` and are separated by `&`; request parameters from a request body can have different forms and can be separated in different ways dependent on the request `Content-Type`, but they, too, have a `name` and a `value`.

All request parameter names and values are strings. A parameter value, however, often designates a value of a specific type, e.g. an integer or a datetime. The response generating code can be simplified significantly when it does not need to make the type conversion itself. In addition, in some cases the request parameters are not independent from one another but related. In those cases it can help if the related parameters are aggregated into a single object. Zope supports both cases but it needs directives to guide the process. It uses `name` suffixes of the form `:directive` to specify such directives. For example, the parameter `i:int=1` tells Zope to convert the value '1' to an integer and use it as value for request variable `i`; the parameter sequence `x.name:record=Peter&x.age:int:record=10` tells Zope to construct a record `x` with attributes `name` and `age` and respective values 'Peter' and 10.

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

The publisher provides a number of additional special variables such as `URL`, `URLn`, `BASEn` and others, which are derived from the request.

Unfortunately, there is no current documentation for those variables.

**Argument Conversion**

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

```python
def one_third(number):
    """returns the number divided by three"""
    return number / 3.0
```
This function cannot be called from the web because by default the publisher marshals arguments into strings, not numbers. This is why the publisher provides a number of converters. To signal an argument conversion you name your form variables with a colon followed by a type conversion code.

For example, to call the above function with 66 as the argument you can use this URL `one_third?number:int=66`.

Some converters employ special logic for the conversion. For example, both `tokens` as well as `lines` convert to a list of strings but `tokens` splits the input at whitespace, `lines` at newlines.

The publisher supports many 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 a Python integer. Also converts a list/tuple of variables to a list/tuple of integers.
- **long** – Converts a variable to a Python integer. Strips the trailing “L” symbol at the end of the value. Also converts a list/tuple of variables to a list/tuple of integers.
- **float** – Converts a variable to a Python floating point number. Also converts a list/tuple of variables to a list/tuple of floats.
- **string** – Converts a variable to a native string. So the result is `str`, no matter which Python version you are on.
- **cstring** – Converts a variable to a Python unicode string.
- **bytes** – Converts a variable to a Python bytes object/string.
- **required** – Raises an exception if the variable is not present or is an empty string.
- **date** – Converts a string to a `DateTime` object. The formats accepted are fairly flexible, for example `10/16/2000, 12:01:13 pm`.
- **date Internacional** – Converts a string to a `DateTime` object, but especially treats ambiguous dates as “days before month before year”. This useful if you need to parse non-US dates.
- **lines** – Converts a variable to a Python list of native strings by splitting the string on line breaks. Also converts list/tuple of variables to list/tuple of native strings.
- **tokens** – Converts a variable to a Python list of native strings by splitting the variable on whitespace.
- **text** – Converts a variable to a native string with normalized line breaks. Different browsers on various platforms encode line endings differently, so this converter makes sure the line endings are consistent, regardless of how they were encoded by the browser.
- **unlines, utokens, utext** – like `lines`, `tokens`, `text`, but always converts into unicode strings.

The full list of supported converters can be found in `ZPublisher.Converters.type_converters`.

If the publisher cannot coerce a request parameter into the type required by the type converter it will raise an error. This is useful for simple applications, but restricts your ability to tailor error messages. If you wish to provide your own error messages, you should convert arguments manually in your published objects rather than relying on the publisher for coercion.

**Note:** Client-side validation with HTML 5 and/or JavaScript may improve the usability of the application, but it is never a replacement for server side validation.

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

An aggregator directive tells Zope how to process parameters with the same or a similar name. Zope supports the following aggregators:

- **list** – collect all values with this name into a list. If there are two or more parameters with the same name they are collected into a list by default. The list aggregator is mainly used to ensure that the parameter leads to a list value even in the case that there is only one of them.

- **tuple** – collect all values with this name into a tuple.

- **default** – use the value of this parameter as a default value; it can be overridden by a parameter of the same name without the default directive.

- **record** – this directive assumes that the parameter name starts with `var.attr`. It tells Zope to create a request variable `var` of type record (more precisely, a `ZPublisher.HTTPRequest.record` instance) and set its attribute `attr` to the parameter value. If such a request variable already exists, then only its attribute `attr` is updated.

- **records** – this directive is similar to record. However, `var` gets as value not a single record but a list of records. Zope starts a new record (and appends it to the list) when the current request parameter would override an attribute in the last record of the list constructed so far (or this list is empty).

- **ignore_empty** – this directive causes Zope to ignore the parameter if its value is empty.

### An aggregator in detail: the record argument

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

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

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

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

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

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

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

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

```html
<input type="hidden"
    name="pizza.toppings:record:list:default"
    value="All">
<select multiple name="pizza.toppings:record:list:ignore_empty">
    <option>Cheese</option>
</select>
```
The default type allows a specified value to be inserted when the form field is left blank. In the above example, if the user does not select values from the list of toppings, the default value will be used. The record pizza will have the attribute toppings and its value will be the list containing the word “All” (if the field is empty) or a list containing the selected toppings.

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

```html
<h2>Member One</h2>
Name:
<input type="text" name="members.name:records"><br>
Email:
<input type="text" name="members.email:records"><br>
Age:
<input type="text" name="members.age:int:records"><br>

<h2>Member Two</h2>
Name:
<input type="text" name="members.name:records"><br>
Email:
<input type="text" name="members.email:records"><br>
Age:
<input type="text" name="members.age:int:records"><br>
```

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

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

**Note:** Records do not work with input fields of type radio as you might expect, as all radio fields with the same name are considered as one group - even if they are in different records. That means, activating one radio button will also deactivate all other radio buttons from the other records.

**Attention:** When using records please note that there is a known issue when you use a form, where checkboxes are used in the first “column”.

As browsers leave out empty checkboxes when sending a request, the object publisher may not be able to match checked checkboxes with the correct record.

This behaviour cannot be fixed.

If you want a checkbox as the first form field, you can work around the problem by using a hidden input field.

**Code example with applied workaround:**

```html
<option>Onions</option>
<option>Anchovies</option>
<option>Olives</option>
<option>Garlic</option>
</select>
```
Specifying argument character encodings

An encoding directive tells the converting process the encoding of the parameter value. Typical encodings are e.g. “utf8” or “latin1”.

An encoding directive is ignored if the parameter does not have a converter directive as well. If there is no encoding directive, the converter uses the default encoding as specified by the Zope configuration option `zpublisher-default-encoding`. The default value for this configuration option in Zope 4 is `utf-8`.

In principle, Zope supports any encoding known by the `codecs` module. However, the converter may impose restrictions.

Special cases

If your pages use a different encoding, such as `Windows-1252` or `ISO-8859-1`, which was the default encoding for HTML 4, you have to add the encoding, e.g. `:cp1252`, for all argument type converts, such as follows:

```
<input type="text" name="name:cp1252:ustring">
<input type="checkbox" name="numbers:list:int:cp1252" value="1">
<input type="checkbox" name="numbers:list:int:cp1252" value="1">
```

Note: For a full list of supported encodings, please have a look at:

https://docs.python.org/3.7/library/codecs.html#standard-encodings

If your pages all use a character encoding which has ASCII as a subset, such as Latin-1, UTF-8, etc., then you do not need to specify any character encoding for boolean, int, long, float and date types.

Note: The form submission encoding can be overridden by the `accept-charset` attribute of the `form` tag:

https://www.w3.org/TR/html5/sec-forms.html#selecting-a-form-submission-encoding

Method Arguments

Normally, a request parameter is transformed into a request variable and made available via the `form` attribute of the request object. The `method` directive tells Zope to extend the path used for traversal.
You can use a **method** directive to control which object is published based on form data. For example, you might want to have a form with a select list that calls different methods depending on the item chosen. Similarly, you might want to have multiple submit buttons which invoke a different method for each button.

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

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

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

**Zope** supports the following method directives: **method** (synonym **action**), and **default_method** (synonym **default_action**). A path extension specified by a **default_method** directive is **overridden** by a **method** directive.

### Processing model for request data marshaling

Zope processes the request parameters in `ZPublisher.HTTPRequest.HTTPRequest.processInputs`.

This section describes the complex processing model in some detail as its various steps and peculiar logic may lead to surprises. If you are developing **with** Zope as opposed to developing **Zope** itself, you may skip over these details.

In a preliminary step the request parameters are collected from the potential sources, i.e. the “query” and request body (if present), and normalized. The result is a sequence of name/value pairs, each describing a single request parameter.

Zope then sets up some variables:

- **form** as target for the collected form variables
- **defaults** as target for the collected form variable defaults
- **tuple_items** to remember which form variable should be tuples
- **method** as target for the path extension from method directives.

It then loops over the request parameter sequence.

For each request parameter, the processing consists of the following steps:

1. Some variables are set up:
   - **isFileUpload** does the parameter represent an uploaded file?
   - **converter_type** the most recently seen converter from a converter directive
   - **character_encoding** the most recently seen encoding from an encoding directive
   - **flags** to indicate which processing types are requested via directives

   Processing types are “ignore”, “aggregate as sequence”, “aggregate as record”, “aggregate as records”, “use as default”, “convert” (using **converter_type** and **character_encoding**).

2. The parameter value is checked to see if it is a file upload. In this case, it is wrapped into a `FileUpload`, and **isFileUpload** is updated.
3. All directives in the parameter name are examined from right to left and the variables set up in step 1 are updated accordingly. :tuple directives update flags and tuple_items, and method directives update flags and method.

4. The actions stored in flags during step 3 are executed.

   If flags indicate the use as default, the step operates on defaults, otherwise on form.

After all request parameters have been processed request variables from defaults are put into form as long as it does not contain that variable already. If a method directive has been encountered the traversal path is extended accordingly.

As a security measure, mainly for DTML use, request variables are not only made available in the request attribute form. A (somewhat) secured version of them is also stored in the attribute taintedform. In the tainted request variable variant, strings potentially containing HTML fragments use TaintedString as data type rather than the normal str. DTML will automatically quote those values to give some protection against cross site scripting attacks via HTML injection. With the more modern page templates, all values (not only tainted ones) are quoted by default. They typically do not use the tainted form of the request variables.

**Known issues and caveats**

1. There is almost no error handling:
   - unrecognized directives are silently ignored
   - if a request parameter contains several converter directives, the leftmost wins
   - if a request parameter contains several encoding directives, the leftmost wins
   - if a request parameter contains an encoding but no converter directive, the encoding directive is silently ignored
   - some directive combinations do not make sense (e.g. :record:records); for them, some of the directives are silently ignored

2. Usually, the order of aggregator directives in a request parameter does not matter. However, this is not the case for the :tuple directive. To really produce a tuple request variable, it must be the left most directive; otherwise, it is equivalent to :list.

   In addition, :tuple is always equivalent to :list for request variables aggregated as record or sequence of records.

3. The main use case for the :default directive is to provide a default value for form controls (e.g. checkboxes) for which the browser may or may not pass on a value when the form is submitted. Unfortunately, this only works at the top level. It does not work for subcomponents, e.g. an attribute of a “record”. As a consequence, if a request parameter combines :default with another aggregator directive, the result may be unexpected.

4. The request preprocessing happens at a very early stage, before traversal has taken place. As a consequence, important configuration for application specific error handling may not yet have taken effect. Exceptions raised during this stage are reported and tracked only via “root level” error handling. For the reason it is typically better to use a form framework such as z3c.form or zope.formlib for form processing rather than the built-in features described in this document.

**Exceptions**

When the object publisher catches an unhandled exception, it tries to match it with a set of predefined exceptions coming from the zExceptions package, such as HTTPNoContent, HTTPNotFound, HTTPUnauthorized.
If there is a match, the exception gets upgraded to the matching `zException`, which then results in a proper response returned to the browser, including an appropriate HTTP status code.

**Note:** For a full list of exceptions please directly refer to the implemented exception classes within the `zExceptions` package.

**Attention:** When you create a custom exception, please make sure not to inherit from `BaseException`, but from `Exception` or one of its child classes, otherwise you’ll run into an exception in `waitress`.

**Note:** Beginning with Zope 4, a standard installation no longer comes with a `standard_error_message`. There are two ways to catch and render an exception:

- create a `standard_error_message`, which can be a DTML Method, DTML Document, Script (Python) or Page Template
- create an exception view, see blog post Catching and rendering exceptions

If the exception is not handled, it travels up the WSGI stack.

What happens then depends entirely on the possibly installed WSGI middleware or the WSGI server. By default Zope uses `waitress` and by default `waitress` returns an error message as follows:

```
Internal Server Error
The server encountered an unexpected internal server error
(generated by waitress)
```

**Note:** Further information:

Debugging Zope applications under WSGI

**Exceptions and Transactions**

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

If an unhandled exception is raised during the publishing process, Zope aborts the transaction. When a `ConflictError` occurs, Zope retries the request up to three times by default. You can change that number in the `zope.conf` by adding a `max_conflict_retries` directive.

**Note:** For further information on transactions please refer to chapter 6 ZODB Persistent Components.
Manual Access to Request and Response

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

```
response=REQUEST.RESPONSE
```

The APIs of request and response can be looked up in the source code.

We’ll look at a few common uses of the request and response. If you need access to the complete API, please directly refer to the source code.

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

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

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

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

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

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

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

```
def calculate(data, REQUEST=None):
  ...
  result = ...
  if REQUEST is not None:
    return "<html><p>Result: $s</p></html>" % result
  return result
```

The `calculate` function can be called from Python, and also from the web. By including `REQUEST=None` in the signature you can differentiate between being called from Python and being called form the web.

### 8.4.3 Other Network Protocols

**XML-RPC**

XML-RPC is a light-weight remote procedure call (RPC) protocol that uses XML to encode its calls and HTTP as a transport mechanism.

All objects in Zope support XML-RPC publishing. Generally you will select a published object as the end-point and select one of its methods as the method. For example you can call the `getId` method on a Zope folder at `http://example.com/myfolder` like so:
from xmlrpc.client import ServerProxy as proxy
folder = proxy("http://example.com/myfolder")
folder_id = folder.getId()

You can also do traversal via a dot notation.

For example:

from xmlrpc.client import ServerProxy as proxy

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

# walking directly up to the published object
folderB = proxy("http://example.com/app/folderA/folderB")
id2 = folderB.getId()
print(id1 == id2)

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

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

**Note:** XML-RPC does not support keyword arguments.

### 8.4.4 Summary

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

### 8.5 Zope Products

**Attention:** This document is currently being reviewed and edited for the upcoming release of Zope 4.

#### 8.5.1 Introduction

In this chapter we are looking at building Python packages that are Zope Products. Products most often provide new addable objects.

**Note:** In the early days of Zope development, it was quite common to develop “through the web”. This is still possible but no longer recommended. For further information please refer to the Zope Book.
8.5.2 Development Process

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

Consider Alternatives

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

Starting with Interfaces

The first step in creating a product is to create one or more interfaces which describe the product. See Chapter 2 for more information on interfaces and how to create them.

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

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

```python
from zope.interface import Interface

class IPoll(Interface):
    """A multiple choice poll""

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

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

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

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

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

How you name your interfaces is entirely up to you. Here we’ve decided to use prefix “I” in the name of the interface.

Implementing Interfaces

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

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

```python
from poll import Poll

class PollImplementation:
```

(continues on next page)
A multiple choice poll, implements the Poll interface.

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

```python
implements(IPoll)

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

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

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

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

def getResponses(self):
    # Returns the sequence of responses
    return tuple(self._responses)

def getQuestion(self):
    # Returns the question
    return self._question
```

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

```python
>>> from PollImplementation import PollImplementation
>>> p = PollImplementation("What's your favorite color?",
...                           ["Red", "Green", "Blue", "I forget"])
...>>> p.getQuestion()
"What's your favorite color?"
...>>> p.getResponses()
('Red', 'Green', 'Blue', 'I forget')
...>>> p.getVotesFor(0)
0
...>>> p.castVote(0)
...>>> p.getVotesFor(0)
1
...>>> p.castVote(2)
...>>> p.getTotalVotes()
2
...>>> p.castVote(4)
```
Interactive testing is one of Python’s great features. It lets you experiment with your code in a simple but powerful way.

At this point you can do a fair amount of work, testing and refining your interfaces and classes which implement them. See Chapter 9 for more information on testing.

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

### Building Product Classes

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

### Base Classes

Consider an example product class definition:

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

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

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

#### Acquisition.Implicit

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

XXX: is this true? I thought that any ExtensionClass.Base can be acquired. The Implicit and Explicit just control how the class can acquire, not how it is acquired.
Globals.Persistent

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

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

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

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

OFS.SimpleItem.Item

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

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

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

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

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

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

```python
class PollProduct(..., Item):
    meta_type = 'Poll'
    ...

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

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

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

OFS.ObjectManager

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

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

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

OFS.PropertyManager

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

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

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

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

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

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

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

Each entry in the _properties structure may optionally provide a mode key, which specifies the mutability of the property. The mode string, if present, must be w, d, or wd.
A w present in the mode string indicates that the value of the property may be changed by the user. A d indicates that the user can delete the property. An empty mode string indicates that the property and its value may be shown in property listings, but that it is read-only and may not be deleted.

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

**Security Declarations**

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

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

```python
from AccessControl import ClassSecurityInfo
class PollProduct(...):
    ...
    security = ClassSecurityInfo()
    security.declareProtected('Use Poll', 'castVote')
    def castVote(self, index):
        ...
    security.declareProtected('View Poll results', 'getTotalVotes')
    def getTotalVotes(self):
        ...
    security.declareProtected('View Poll results', 'getVotesFor')
    def getVotesFor(self, index):
        ...
    security.declarePublic('getResponses')
    def getResponses(self):
        ...
    security.declarePublic('getQuestion')
    def getQuestion(self):
        ...
```

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

```python
from Globals import InitializeClass
class PollProduct(...):
    ...
InitializeClass(PollProduct)
```

**Summary**

Congratulations, you’ve created a product class. Here it is in all its glory (see examples/PollProduct.py):
from `Poll` import `Poll`
from `AccessControl` import `ClassSecurityInfo`
from `Globals` import `InitializeClass`
from `Acquisition` import `Implicit`
from `Globals` import `Persistent`
from `AccessControl.Role` import `RoleManager`
from `OFS.SimpleItem` import `Item`

class `PollProduct`(`Implicit`, `Persistent`, `RoleManager`, `Item`):
    
    """Poll product class, implements Poll interface.
    
The poll has a question and a sequence of responses. Votes are stored in a dictionary which maps response indexes to a number of votes.
    """

    implements(`IPoll`)

    `meta_type` = 'Poll'

    `security` = `ClassSecurityInfo`

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

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

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

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

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

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

(continues on next page)
Now it’s time to test your product class in Zope. To do this you must register your product class with Zope.

### 8.5.3 Registering Products

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

#### Product Initialization

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

```python
from PollProduct import PollProduct, addForm, addFunction

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

This function makes one call to the registrar object which registers a class as an addable object. The registrar figures out the name to put in the product add list by looking at the ‘meta_type’ of the class. Zope also deduces a permission based on the class’s meta-type, in this case Add Polls (Zope automatically pluralizes “Poll” by adding an “s”). The ‘constructors’ argument is a tuple of objects consisting of two functions: an add form which is called when a user selects the object from the product add list, and the add method which is the method called by the add form. Note that these functions are protected by the constructor permission.

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

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

#### Factories and Constructors

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

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

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

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

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

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

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

Here’s an add function for your poll class:

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

The dispatcher has three methods:

- `Destination` – The `ObjectManager` where your product was added.
- `DestinationURL` – The URL of the `ObjectManager` where your product was added.
- `manage_main` – Redirects to a management view of the `ObjectManager` where your product was added.

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

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

Finally you should recognize that the constructor functions are *not* methods on your product class. In fact they are called before any instances of your product class are created. The constructor functions are published on the web so they need to have doc strings, and are protected by a permission defined in during product initialization.

**Testing**

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

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

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

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

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

poll = context

return float(poll.getVotesFor(index)) / poll.getTotalVotes()
```

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

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

<form> <!-- calls this dtml method -->
  <dtml-in getResponses>
    <p>
      <input type="radio" name="index" value="&dtml-sequence-index;"> 
      <dtml-var sequence-item>
    </p>
  </dtml-in>
<input type="submit" value=" Vote ">
</form>

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

<!-- display results -->
<h2>Results</h2>
<p><dtml-var getTotalVotes> votes cast</p>
<dtml-in getResponses>
  <p>
    <dtml-var sequence-item> - 
    <dtml-var expr="getPercentFor(_.get('sequence-index'))">%
  </p>
</dtml-in>
```

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

At this point there’s quite a bit of testing and refinement that you can do. Your main annoyance will be having to restart Zope each time you make a change to your product class (but see Chapter 9 for information on how to avoid all this restarting). If you vastly change your class you may break existing poll instances, and will need to delete them and create new ones. See Chapter 9 for more information on debugging techniques which will come in handy.

### Building Management Interfaces

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

### Defining Management Views

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

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

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

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

- *‘label’* – This is the name of the management view
- *‘action’* – This is the URL that is called when the view is chosen. Normally this is the name of a method that displays a management view.
- *‘target’* – An optional target frame to display the action. This item is rarely needed.
- *‘help’* – Optional help information associated with the view. You’ll find out more about this option later.

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

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

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

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

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

Creating Management Views

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

```python
from Globals import DTMLFile
class PollProduct(...):
    ...
    editForm = DTMLFile('dtml/edit', globals())
    ...
```

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

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

```python
from Globals import DTMLFile
class PollProduct(...):
    ...
    _editForm = DTMLFile('dtml/edit', globals())

    def editForm(self, ...):
        ...
        return self._editForm(REQUEST, ...)
```

When creating management views you should include the DTML variables manage_page_header and manage_tabs at the top, and manage_page_footer at the bottom. These variables are acquired by your product and draw a standard management view header, tabs widgets, and footer. The management header also includes CSS information which you can take advantage of. You can use any of the styles Bootstrap 4 provides. (See http://getbootstrap.com/docs/4.1/)

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

```html
<dtml-var manage_page_header>
<dtml-var manage_tabs>
```
This form allows you to change the poll’s question and responses. Changing a poll’s question and responses will reset the poll’s vote tally.

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

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

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

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

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

```python
class PollProduct(...):
    ...
```
Notice how the \_init\_ method has been refactored to use the new editPoll method. Also notice how the editPoll method is protected by a new permissions, Change Poll.

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

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

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

If this method is called from the web, then Zope will automatically supply the REQUEST parameter. (See chapter 4 for more information on object publishing). By testing the REQUEST you can find out if your method was called from the web or not. If you were called from the web you return the edit form again.

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

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

```python
def editPoll(self, question, responses, REQUEST=None):
    """Changes the question and responses.""
    ...
Now let's take a look at how to define an icon for your product.

**Icons**

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

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

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

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

**Online Help**

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

**Context Sensitive Help**

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Other User Interfaces

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

FTP and WebDAV Interfaces

Both FTP and WebDAV treat Zope objects like files and directories. See Chapter 3 for more information on FTP and WebDAV.

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

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

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

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

Content Management Framework Interface

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

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

Packaging Products

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

```
$ tar zcvf MyProduct-1.0.1.tgz MyProduct
```

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

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

Product Information Files

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

- **README.txt** – Provides basic information about your product. Zope will parse this file as StructuredText and make it available on the README view of your product in the control panel.

- **VERSION.txt** – Contains the name and version of your product on a single line. For example, ‘Multiple Choice Poll 1.1.0’. Zope will display this information as the ‘version’ property of your product in the control panel.

- **LICENSE.txt** – Contains your product license, or a link to it.

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

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

- **TODO.txt** – This file should make clear where this product release needs work, and what the product author intends to do about it.
• **CHANGES.txt** and **HISTORY.txt** – ‘CHANGES.txt’ should enumerate changes made in particular product versions from the last release of the product. Optionally, a ‘HISTORY.txt’ file can be used for older changes, while ‘CHANGES.txt’ lists only recent changes.

• **DEPENDENCIES.txt** – Lists dependencies including required os platform, required Python version, required Zope version, required Python packages, and required Zope products.

**Product Directory Layout**

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

- **www** – Contains your icon & ZPT files.
- **help** – Contains your help files.
- **tests** – Contains your unit tests.

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

**8.5.4 Evolving Products**

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

**Evolving Classes**

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

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

```python
class Sandwich(...):
    improved_spam = None
    ...

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

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

A third option is to create a method to update old instances. Then you can manually call this method on instances to update to them. Note, this won’t work unless the instances function well enough to be accessible via the Zope management screens.
While you are developing a product you won’t have to worry too much about these details, since you can always delete old instances that break with new class definitions. However, once you release your product and other people start using it, then you need to start planning for the eventuality of upgrading.

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

Evolving Interfaces

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

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

8.5.5 Conclusion

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

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

8.6 ZODB Persistent Components

**Attention:** This document was written for Zope 2.

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

8.6.1 Persistent Objects

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

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

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

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

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

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

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

After installing ZODB, you can start to experiment with it right from the Python command line interpreter. If you’ve installed Zope, before running this set of commands, shut down your Zope server, and “cd” to the “lib/python” directory of your Zope instance. If you’re using a “standalone” version of ZODB, you likely don’t need to do this, and you’ll be able to use ZODB by importing it from a standard Python package directory. In either case, try the following set of commands:

```
chrism@saints:/opt/zope/lib/python$ python
Python 2.1.1 (#1, Aug 8 2001, 21:17:50)
[GCC 2.95.2 20000220 (Debian GNU/Linux)] on linux2
Type "copyright", "credits" or "license" for more information.
>>> from ZODB import FileStorage, DB
>>> storage = FileStorage.FileStorage('mydatabase.fs')
>>> db = DB(storage)
>>> connection = db.open()
>>> root = connection.root()
```

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

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

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

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

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

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

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

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

Working with simple python types is useful, but the real power of ZODB comes out when you store your own kinds of objects in the database. For example, consider a class that represents a employee:
Calling ‘setName’ will set a name for the employee. Now, you can put Employee objects in your database:

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

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

### 8.6.2 Persistent Rules

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

- Your objects, and their attributes, must be “pickleable”.
- Your object cannot have any attributes that begin with ‘_p_’.
- Attributes of your object that begin with ‘_v_’ are “volatile” and are not saved to the database (see next section).
- You must explicitly signal any changes made to mutable attributes (such as instances, lists, and dictionaries) or use persistent versions of mutable objects, like ‘ZODB.PersistentMapping’ (see below for more information on ‘PersistentMapping’).

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

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

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

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

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

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

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

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

---

8.6. ZODB Persistent Components 463
class DepartmentManager(Persistent):

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

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

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

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

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

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

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

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

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

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

### 8.6.3 Transactions and Persistent Objects

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

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

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

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

5. The transaction commits

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

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

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

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

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

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

### Subtransactions

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

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

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

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

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

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

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

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

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

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

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

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

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

Resolving Conflicts

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

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

```python
class Counter(Persistent):
    self.count = 0

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

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

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

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

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

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

‘newState’ – The state after changes made by the current transaction. The method is not permitted to modify this value. This method should compute a new state by merging changes reflected in ‘savedState’ and ‘newState’, relative to ‘oldState’.
The method should return the state of the object after resolving the differences.

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

```python
class Counter(Persistent):
    self.count = 0

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

    def _p_resolveConflict(self, oldState, savedState, newState):
        # Figure out how each state is different:
        savedDiff = savedState['count'] - oldState['count']
        newDiff = newState['count'] - oldState['count']

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

        return oldState
```

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

### 8.6.4 Threadsafety of Non-Persistent Objects

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

#### Mutable Default Arguments

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

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

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

- Don’t use mutable default arguments. (Good)
- If you use them, you cannot change them. If you want to change them, you will need to implement your own locking. (Bad)

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

#### Shared Module Data

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

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

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

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

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

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

```python
class Queue(Persistent):
    def __init__(self):
        self.list=[]

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

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

Notice how this class uses the mutable object `self.list` immutably. If this class used `self.list.pop` and `self.list.append`, then the persistence machinery would not notice that `self.list` had changed.

### 8.6.5 Shared External Resources

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

### 8.6.6 Other ZODB Resources

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

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

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

8.7 Acquisition

**Attention:** This document was written for Zope 2.

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

8.7.1 Introductory Example

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

```python
import ExtensionClass, Acquisition

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

class A(Acquisition.Implicit):
    
    def report(self):
        print self.color

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

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

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

d.a.report() # prints 'green'
a.report() # raises an attribute error
```

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

8.7.2 Acquisition Wrappers

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

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

<table>
<thead>
<tr>
<th>'c.a.aq_parent is c'</th>
</tr>
</thead>
</table>

and:

<table>
<thead>
<tr>
<th>'c.a.aq_self is a'</th>
</tr>
</thead>
</table>

both evaluate to true, but the expression:

<table>
<thead>
<tr>
<th>'c.a is a'</th>
</tr>
</thead>
</table>

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

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

You can manually wrap an instance of an object that inherits from an acquisition base class by using its ‘__of__’ method. For example:

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

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

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

The expression ‘b.__of__(a)’ wraps ‘b’ in an acquisition wrapper explicitly, and returns the acquisition wrapper. The ‘color’ attribute of ‘a’ is found via acquisition when this expression is executed.

### 8.7.3 Explicit and Implicit Acquisition

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

**Implicit acquisition**

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

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

To support implicit acquisition, your class should inherit from the mix-in class ‘Acquisition.Implicit’.

**Explicit Acquisition**

When explicit acquisition is used, attributes are not automatically obtained from the environment. Instead, the method ‘aq_acquire’ must be used. For example:
To support explicit acquisition, your class should inherit from the mix-in class ‘Acquisition.Explicit’.

### 8.7.4 Controlling Acquisition

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

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

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

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

### 8.7.5 Filtered Acquisition

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

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

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

For example, in:

```python
from Acquisition import Explicit

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

class E(Explicit, HandyForTesting): pass
```

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

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

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

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

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

spam(Nice) and I am nice!

Filtered acquisition is rarely used in Zope.

8.7.6 Acquiring from Context

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

Most of the example’s we’ve seen so far show establishing of an acquisition context using ‘getattr’ symantactics. For
example, ‘a.b’ is a reference to ‘b’ in the context of ‘a’.

You can also manually set acquisition context using the ‘__of__’ method. For example:

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

In this case, ‘a’ does not contain ‘b’, but it is put in ‘b”s context using the ‘__of__’ method.

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

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

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

(continues on next page)
print a.b.x.color # prints red

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

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

### 8.7.7 Containment Before Context

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

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

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

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

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

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

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

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

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

### 8.7.8 Additional Attributes and Methods

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

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

is equivalent to:

```python
a.x
```

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

```python
a.b.x.aq_chain # returns [x, b, a]
```
You can find out if an object is in the acquisition context of another object using the `aq_inContextOf` method. For example:

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

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

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

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

**Acquisition Module Functions**

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

‘aq_acquire(object, name [, filter, extra, explicit, default, containment])’ – Acquires an object with the given name.

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

The function accepts a number of optional arguments:

- **‘filter’** – A callable filter object that is used to decide if an object should be acquired.

  The filter is called with five arguments:
  
  - The object that the aq_acquire method was called on,
  - The object where an object was found,
  - The name of the object, as passed to aq_acquire,
  - The object found, and
  - The extra argument passed to aq_acquire.

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

- **‘extra’** – extra data to be passed as the last argument to the filter.

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

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

- **‘default’** – A default value to return if no value can be acquired.

- **‘containment’** – A flag indicating whether the search should be limited to the containment hierarchy.

In addition, arguments can be provided as keywords.

- **‘aq_base(object)’** – Return the object with all wrapping removed.
• ‘aq_chain(object [, containment])’ – Return a list containing the object and its acquisition parents. The optional argument, ‘containment’, controls whether the containment or access hierarchy is used.

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

• ‘aq_inner(object)’ – Return the object with all but the innermost layer of wrapping removed.

• ‘aq_parent(object)’ – Return the acquisition parent of the object or ‘None’ if the object is unwrapped.

• ‘aq_self(object)’ – Return the object with one layer of wrapping removed, unless the object is unwrapped, in which case the object is returned.

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

8.7.9 Acquisition and Methods

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

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

8.7.10 Conclusion

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

8.8 Security

8.8.1 Introduction

A typical web application needs to be securely managed. Different types of users need different kinds of access to the components that make up an application. To this end, Zope includes a comprehensive set of security features. This chapter’s goal is to shed light on Zope security in the context of Zope Product development. For a more fundamental overview of Zope security, you may wish to refer to the Zope Book Chapter “Users and Security” <https://zope.readthedocs.io/en/latest/zopebook/Security.html>_ Before diving into this chapter, you should have a basic understanding of how to build Zope Products as well as an understanding of how the Zope object publisher works. This is covered in Object Publishing.

8.8.2 Security Architecture

The Zope security architecture is built around a security policy, which you can think of as the “access control philosophy” of Zope. This policy arbitrates the decisions Zope makes about whether to allow or deny access to any particular object defined within the system.
How The Security Policy Relates to Zope’s Publishing Machinery

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

How The Security Policy Relates to Restricted Code

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

When restricted code runs, any access to objects integrated with Zope security is arbitrated by the security policy. For example if you write a bit of restricted code with a line that attempts to manipulate an object you don’t have sufficient permission to use, the security policy will deny access to the object. This is accomplished by raising an Unauthorized exception, which is a Python exception caught by the publisher and handed to a user folder, which will then attempt to get user credentials before continuing with the request. The particular code used to attempt to obtain the credentials is determined by the User Folder “closest” (folder-wise) to the object being accessed.

Unauthorized Exceptions and Through-The-Web Code

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

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

How The Security Policy Relates To Unrestricted Code

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

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

Details Of The Default Zope Security Policy

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

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

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

**Overview Of Using Zope Security Within Your Product**

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

For a more fundamental overview of Zope users, roles, and permissions, see the section titled “Authorization, Roles and Permissions” in the Security Chapter of the Zope Book.

**Security Declarations In Zope Products**

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

• public – allow anybody to access the protected object or method
• private – deny anyone access to the protected object or method
• protected – protect access to the object or method with a permission

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

**8.8.3 Permissions In Zope Products**

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

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

It is important to note that Zope’s security architecture dictates that roles and users remain the domain of administrators, while permissions remain the domain of developers. Developers of Products should not attempt to define roles or users, although they may (and usually must) define permissions. Most importantly, a Zope administrator who makes use of your product should have the “last word” as regards which roles are granted which permissions, allowing her to protect her site in a manner that fits her business goals.
Permission names are strings, and these strings are currently arbitrary. There is no permission hierarchy, or list of “approved permissions”. Developers are encouraged to reuse Zope core permissions (e.g. “View”, “Access contents information”) where appropriate, or they may create their own as the need arises. It is generally wise to reuse existing Zope permission names unless you specifically need to define your own. For a list of existing Zope core permissions, see Appendix A: Zope Core Permissions.

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

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

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

### 8.8.4 Implementing Security In Python Products

**Security Assertions**

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

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

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

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

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

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

Making Security Assertions

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

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

8.8.5 Class Security Assertions

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

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

Declaring Class Security

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

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

The `ClassSecurityInfo` object can be used to declare access in two ways, as a function decorator or by calling the required method explicitly.

For example:

```python
from AccessControl import ClassSecurityInfo
class Mailbox(ObjectManager):
    """A mailbox object that contains mail message objects.""

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

    # Here is an example of a security assertion using a decorator.
    # We are declaring that access to messageCount is public.
    @security.public
```

(continues on next page)
def messageCount(self):
    """Return a count of messages."""
    return len(self._messages)

In the example above we decorated the `messageCount` method with the decorator method `security.public` of the `ClassSecurityInfo` instance to declare that access to the `messageCount` method be public. To make security assertions for your object, you just call the appropriate methods of the `ClassSecurityInfo` object, passing the appropriate information for the assertion you are making.

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

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

To assert that a method is `public` (anyone may call it) you may use the `public` decorator:

```python
@security.public
def myMethod(self):
    ...
```

To assert that a method is `private` you may use the `private` decorator:

```python
@security.private
def myMethod(self):
    ...
```

To assert that a method or subobject is `protected` by a particular permission, you use the `protected` decorator, passing a permission name:

```python
@security.protected(permissionName)
def myMethod(self):
    ...
```

If you have lots of methods you want to protect under the same permission, you can pass as many methodNames as you want to a call to the `declareProtected` method:

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

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

**Deciding To Use Protected vs. Public or Private**

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

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

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

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

```python
from AccessControl import ClassSecurityInfo
from AccessControl.class_init import InitializeClass

class Mailbox(ObjectManager):
    """A mailbox object.""
    # Create a SecurityInfo for this class
    security = ClassSecurityInfo()
    security.declareProtected('View management screens', 'manage')
    manage = HTMLFile('mailbox_manage', globals())

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

    @security.protected('View Mailbox')
    def listMessages(self):
        """Return a sequence of message objects.""
        return self._messages[:]

    @security.private
    def getMessages(self):
        self._messages=GoGetEm()
        return self._messages

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

Note the last line in the example. In order for security assertions to be correctly applied to your class, you must call the global class initializer `InitializeClass` for all classes that have security information. This is very important - the global initializer does the “dirty work” required to ensure that your object is protected correctly based on the security assertions that you have made. If you don’t run it on the classes that you’ve protected with security assertions, the security assertions will not be effective.

Deciding Permission Names For Protected Methods

When possible, you should make use of an existing Zope permission within protected/declareProtected assertions. A list of the permissions which are available in a default Zope installation is available within Appendix A: Zope Core Permissions. When it’s not possible to reuse an existing permission, you should choose a permission name which is a verb or a verb phrase.

Object Assertions

Often you will also want to make a security assertion on the object itself. This is important for cases where your objects may be accessed in a restricted environment such as a Page Template or a Python Script. Consider the example Page Template code:
Here we are trying to call `some_method`, passing the object `someObject`. When this is evaluated in the restricted environment, the security policy will attempt to validate access to both `some_method` and `someObject`. We've seen how to make assertions on methods - but in the case of `someObject` we are not trying to access any particular method, but rather the **object itself** (to pass it to `some_method`). Because the security machinery will try to validate access to `someObject`, we need a way to let the security machinery know how to handle access to the object itself in addition to protecting its methods.

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

```python
security.declareObjectPublic()
security.declareObjectPrivate()
security.declareObjectProtected(permissionName)
```

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

**An Object Assertion Example**

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

```python
from AccessControl import ClassSecurityInfo
from AccessControl.class_init import InitializeClass

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

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

    # Set security for the object itself
    security.declareObjectProtected('View Mailbox')
    security.declareProtected('View management screens', 'manage')
    manage=HTMLFile('mailbox_manage', globals())

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

    # protect 'listMessages' with the 'View Mailbox' permission
    @security.protected('View Mailbox')
    def listMessages(self):
        """Return a sequence of message objects.""
        return self._messages[:]

    @security.private
    def getMessages(self):
        self._messages=GoGetEm()
        return self._messages
```

(continues on next page)
# call this to initialize framework classes, which does the right thing with the security assertions.
InitializeClass(Mailbox)

## Other Assertions

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

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

```python
security.setDefaultAccess('allow')
```

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

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

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

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

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

```python
class Foo(SimpleItem):
    security = ClassSecurityInfo()
    meta_type = 'Foo'

    @security.protected('View foos')
    def index_html(self):
        """ make index_html web-publishable ""
        return '<html><body>hi!</body></html>'

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

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

Note that Zope will *not* warn you if you misspell the name of a method in a declareProtected, declarePublic, or declarePrivate call. For instance, you try to protect the `index_html` method with the View permission and make a mistake, spelling the name `index_html` as `inde_html`, like so:

```python
8.8. Security
```
security.declareProtected('View', 'inde_html')
# whoops, declared a permission assertion for 'inde_html'
# when I really wanted it to be 'index_html'!
def index_html(self):
    """make index_html web-publishable """
    return '<html><body>hi!</body></html>'

You’ll need to track down these kinds of problems yourself.

## Setting Default Roles For Permissions

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

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

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

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

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

```
security.setPermissionDefault(permissionName, rolesList)
```

The `permissionName` argument should be the name of a permission that you have used in your object and `rolesList` should be a sequence (tuple or list) of role names that should be associated with `permissionName` by default on instances of your object.

Note that it is not always necessary to use this method. All permissions for which you did not set defaults using `setPermissionDefault` are assumed to have a single default role of Manager. Notable exceptions to this rule include View and Access contents information, which always have the default roles Manager and Anonymous.

The `setPermissionDefault` method of the SecurityInfo object should be called only once for any given permission name.

### An Example of Associating Default Roles With Permissions

Here is our Mailbox example, updated to associate the View Mailbox permission with the roles Manager and Mailbox Owner by default:

```python
from AccessControl import ClassSecurityInfo
from AccessControl.class_init import InitializeClass

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

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

    (continues on next page)
```
# Set security for the object itself
security.declareObjectProtected('View Mailbox')

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

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

@security.protected('View Mailbox')
def listMessages(self):
    """Return a sequence of message objects."""
    return self._messages[:]

security.setPermissionDefault('View Mailbox', ('Manager', 'Mailbox Owner'))

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

What Happens When You Make A Mistake Declaring Default Roles?

It’s possible that you will make a mistake when making default roles declarations. For example, it is not legal to declare two conflicting default roles for a permission:

class Foo(SimpleItem):
    security = ClassSecurityInfo()
    meta_type = 'Foo'

    @security.protected('View foos')
def index_html(self):
        """"
        return '<html><body>hi!</body></html>'

security.setPermissionDefault('View foos', ('Manager',))
security.setPermissionDefault('View foos', ('Anonymous',))
# whoops, conflicting permission defaults!

When you make a mistake like this, the security machinery will accept the first declaration made in the code and will write an error to the Zope debug log about the second and following conflicting declarations upon class initialization.

What Can (And Cannot) Be Protected By Class Security Info?

It is important to note what can and cannot be protected using the ClassSecurityInfo interface. First, the security policy relies on Acquisition to aggregate access control information, so any class that needs to work in the security policy must have either Acquisition.Implicit or Acquisition.Explicit in its base class hierarchy.

The current security policy supports protection of methods and protection of subobjects that are instances. It does not currently support protection of simple attributes of basic Python types like string, int, list or dict. For

8.8. Security 485
instance:

```python
from AccessControl import ClassSecurityInfo
from OFS.ObjectManager import ObjectManager

# We subclass ObjectManager, which has Acquisition in its
# base class hierarchy, so we can use SecurityInfo.
class MyClass(ObjectManager):
    """example class""

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

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

    # This is ok, because subObject is an instance
    security.declareProtected('View management screens', 'subObject')
    subObject = MySubObject()

    # This is ok, because sayHello is a method
    @security.public
    def sayHello(self):
        """Return a greeting.""
        return 'hello!'

    # This will not work, because foobar is not a method
    # or an instance - it is a standard Python type
    security.declarePublic('foobar')
    foobar = 'some string'
```

Keep this in mind when designing your classes. If you need simple attributes of your objects to be accessible (say via TAL or DTML), then you need to use the setDefaultAccess method of SecurityInfo in your class to allow this (see the note above about the security implications of this). In general, it is always best to expose the functionality of your objects through methods rather than exposing attributes directly.

Note also that the actual ClassSecurityInfo instance you use to make security assertions is implemented such that it is never accessible from restricted code or through the Web, so no action on the part of the programmer is required to protect it.

### Inheritance And Class Security Declarations

Python inheritance can prove confusing in the face of security declarations.

If a base class which has already been run through InitializeClass is inherited by a subclass, nothing special needs to be done to protect the base class’ methods within the subclass unless you wish to modify the declarations made in the base class. The security declarations “filter down” into the subclass.

On the other hand, if a base class hasn’t been run through the global class initializer (InitializeClass), you need to proxy its security declarations in the subclass if you wish to access any of its methods within through-the-web code or via URL traversal.

In other words, security declarations that you make using ClassSecurityInfo objects effect instances of the class upon which you make the declaration. You only need to make security declarations for the methods and subobjects that your class actually defines. If your class inherits from other classes, the methods of the base classes are protected by the security declarations made in the base classes themselves. The only time you would need to make a security
declaration about an object defined by a base class is if you needed to redefine the security information in a base class for instances of your own class. An example below redefines a security assertion in a subclass:

```
from AccessControl import ClassSecurityInfo
from AccessControl.class_init import InitializeClass
from OFS.ObjectManager import ObjectManager

class MailboxBase(ObjectManager):
    """A mailbox base class."""

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

    @security.protected('View Mailbox')
    def listMessages(self):
        """Return a sequence of message objects."""
        return self._messages[:]

    security.setPermissionDefault('View Mailbox',
                                 ('Manager', 'Mailbox Owner'))

InitializeClass(MailboxBase)

class MyMailbox(MailboxBase):
    """A mailbox subclass

    Here we want the security for listMessages to be public instead of
    protected (as defined in the base class).
    """

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

    security.declarePublic('listMessages')

InitializeClass(MyMailbox)
```

**Class Security Assertions In Non-Product Code (External Methods/Python Scripts)**

---

**Note:** The examples in this section use so-called “External Methods”, which require installing the Products.ExternalMethod package. Use of that package is discouraged in favor of filesystem-based Product code.

Objects that are returned from Python Scripts or External Methods need to have assertions declared for themselves before they can be used in restricted code. For example, assume you have an External Method that returns instances of a custom `Book` class. If you want to call this External Method from a Page Template, and you’d like your template to be able to use the returned `Book` instances, you will need to ensure that your class supports `Acquisition`, and you’ll need to make security assertions on the `Book` class and initialize it with the global class initializer (just as you would with a class defined in a Product). For example:

```python
# an external method that returns Book instances
from AccessControl import ClassSecurityInfo
from AccessControl.class_init import InitializeClass
```

(continues on next page)
from Acquisition import Implicit

class Book(Implicit):
    def __init__(self, title):
        self._title = title

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

    @security.public
    def getTitle(self):
        return self._title

InitializeClass(Book)

# The actual external method

def getBooks(self):
    books = []
    books.append(Book('King Lear').__of__(self))
    books.append(Book('Romeo and Juliet').__of__(self))
    books.append(Book('The Tempest').__of__(self))
    return books

Note that we wrap the book instances by way of their __of__ methods to obtain a security context before returning them.

Note that this particular example is slightly dangerous. You need to be careful that classes defined in external methods not be made persistent, as this can cause Zope object database inconsistencies. In terms of this example, this would mean that you would need to be careful to not attach the Book object returned from the getBooks method to a persistent object within the ZODB. See ZODB Persistent Components for more information. Thus it’s generally a good idea to define the Book class in a Product if you want books to be persistent. It’s also less confusing to have all of your security declarations in Products.

However, one benefit of the SecurityInfo approach is that it is relatively easy to subclass and add security assertions to classes that you did not write. For example, in an External Method, you may want to return instances of Book although Book is defined in another module out of your direct control. You can still use SecurityInfo to define security information for the class by using:

# an external method that returns Book instances

from AccessControl import ClassSecurityInfo
from AccessControl.class_init import InitializeClass
from Acquisition import Implicit
import bookstuff

class Book(Implicit, bookstuff.Book):
    security = ClassSecurityInfo()
    security.declareObjectPublic()
    security.declarePublic('getTitle')

InitializeClass(Book)

# The actual external method
def getBooks(self):
    books=[]
    books.append(Book('King Lear'))
    books.append(Book('Romeo and Juliet'))
    books.append(Book('The Tempest'))
    return books

8.8.6 Module Security Assertions

Another kind of SecurityInfo object you will use as a component developer is the ModuleSecurityInfo object.

ModuleSecurityInfo objects do for objects defined in modules what ClassSecurityInfo objects do for methods defined in classes. They allow module-level objects (generally functions) to be protected by security assertions. This is most useful when attempting to allow through-the-web code to import objects defined in a Python module.

One major difference between ModuleSecurityInfo objects and ClassSecurityInfo objects is that ModuleSecurityInfo objects cannot be declared protected by a permission. Instead, ModuleSecurityInfo objects may only declare that an object is public or private. This is due to the fact that modules are essentially “placeless”, global things, while permission protection depends heavily on “place” within Zope.

Declaring Module Security

In order to use a filesystem Python module from restricted code such as Python Scripts, the module must have Zope security declarations associated with functions within it. There are a number of ways to make these declarations:

- By embedding the security declarations in the target module. A module that is written specifically for Zope may do so, whereas a module not specifically written for Zope may not be able to do so.

- By creating a wrapper module and embedding security declarations within it. In many cases it is difficult, impossible, or simply undesirable to edit the target module. If the number of objects in the module that you want to protect or make public is small, you may wish to simply create a wrapper module. The wrapper module imports objects from the wrapped module and provides security declarations for them.

- By placing security declarations in a filesystem Product. Filesystem Python code, such as the __init__.py of a Product, can make security declarations on behalf of an external module. This is also known as an “external” module security info declaration.

The ModuleSecurityInfo class is defined in the AccessControl package of the Zope framework.

Using ModuleSecurityInfo Objects

Instances of ModuleSecurityInfo are used in two different situations. In embedded declarations, inside the module they affect. And in external declarations, made on behalf of a module which may never be imported.

Embedded ModuleSecurityInfo Declarations

An embedded ModuleSecurityInfo declaration causes an object in its module to be importable by through-the-web code.

Here’s an example of an embedded declaration:
from AccessControl import ModuleSecurityInfo
modulesecurity = ModuleSecurityInfo()
modulesecurity.declarePublic('foo')

def foo():
    return 'hello'
# foo
modulesecurity.apply(globals())

When making embedded ModuleSecurityInfo declarations, you should instantiate a ModuleSecurityInfo object and assign it to a name. It’s wise to use the recommended name `modulesecurity` for consistency’s sake. You may then use the `modulesecurity` object’s `declarePublic` method to declare functions inside of the current module as `public`. Finally, appending the last line (`modulesecurity.apply(globals())`) is an important step. It’s necessary in order to poke the security machinery into action. The above example declares the `foo` function public.

The name `modulesecurity` is used for consistency and for the benefit of new component authors, who often learn from looking at other people’s code. You do not have to use the name `modulesecurity` for the security infrastructure to recognize your assertion information, but it is recommended as a convention.

External ModuleSecurityInfo Declarations

By creating a ModuleSecurityInfo instance with a module name argument, you can make declarations on behalf of a module without having to edit or import the module.

Here’s an example of an external declaration:

```python
from AccessControl import ModuleSecurityInfo
# protect the 'foo' function within (yet-to-be-imported) 'foomodule'
ModuleSecurityInfo('foomodule').declarePublic('foo')
```

This declaration will cause the following code to work within Python Scripts:

```python
from foomodule import foo
```

When making external ModuleSecurityInfo declarations, you needn’t use the `modulesecurity.apply(globals())` idiom demonstrated in the embedded declaration section above. As a result, you needn’t assign the ModuleSecurityInfo object to the name `modulesecurity`.

Providing Access To A Module Contained In A Package

Note that if you want to provide access to a module inside of a package which lives in your PYTHONPATH, you’ll need to provide security declarations for all of the the packages and sub-packages along the path used to access the module.

For example, assume you have a function `foo`, which lives inside a module named `module`, which lives inside a package named `package2`, which lives inside a package named `package1` You might declare the `foo` function public via this chain of declarations:

```python
ModuleSecurityInfo('package1').declarePublic('package2')
ModuleSecurityInfo('package1.package2').declarePublic('module')
ModuleSecurityInfo('package1.package2.module').declarePublic('foo')
```

Note that in the code above we took the following steps:
• make a ModuleSecurityInfo object for package1

• call the declarePublic method of the package1 ModuleSecurityInfo object, specifying package2 as what we’re declaring public. This allows through the web code to “see” package2 inside package1.

• make a ModuleSecurityInfo object for package1.package2.

• call the declarePublic method of the package1.package2 ModuleSecurityInfo object, specifying module as what we’re declaring public. This allows through the web code to “see” package1.package2.module.

• declare foo public inside the ModuleSecurityInfo for package1.package2.module.

Through-the-web code may now perform an import ala: import package1.package2.module.foo

**Declaring Module Security On Modules Implemented In C**

Certain modules, such as the standard Python sha module, provide extension types instead of classes, as the sha module is implemented in C. Security declarations typically cannot be added to extension types, so the only way to use this sort of module is to write a Python wrapper class, or use External Methods.

**Default Module Security Info Declarations**

Through-the-web Python Scripts are by default able to import a small number of Python modules for which there are security declarations. These include string, math, and random. The only way to make other Python modules available for import is to add security declarations to them in the filesystem.

**Utility Functions For Allowing Import of Modules By Through The Web Code**

Instead of manually providing security declarations for each function in a module, the utility function allow_class and allow_module have been created to help you declare the entire contents of a class or module as public.

You can handle a module, such as base64, that contains only safe functions by writing allow_module('module_name'). For instance:

```python
from Products.PythonScripts.Utility import allow_module
allow_module('base64')
```

This statement declares all functions in the base64 module (encode, decode, encodestring, and decodestring) as public, and from a script you will now be able to perform an import statement such as from base64 import encodestring.

To allow access to only some names in a module, you can eschew the allow_class and allow_module functions for the lessons you learned in the previous section and do the protection “manually”:

```python
from AccessControl import ModuleSecurityInfo
ModuleSecurityInfo('module_name').declarePublic('name1','name2', ...)
```

**Making Permission Assertions On A Constructor**

When you develop a Python filesystem-based product, you will generally be required to make “constructor” methods for the objects which you wish to make accessible via the Zope management interface by users of your Product. These constructors are usually defined within the modules which contain classes which are intended to be turned into Zope instances. For more information on how constructors are used in Zope with security, see the chapter Zope Products.
The Zope Product machinery “bootstraps” Product-based classes with proper constructors into the namespace of the Zope management interface *Add* list at Zope startup time. This is done as a consequence of registering a class by way of the Product’s __init__.py initialize function. If you want to make, for example, the imaginary *FooClass* in your Product available from the *Add* list, you may construct an __init__.py file that looks much like this:

```python
from FooProduct import FooClass

def initialize(context):
    """ Initialize classes in the FooProduct module """
    context.registerClass(
        FooProduct.FooClass,  # the class object
        permission='Add FooClasses',
        constructors=(FooProduct.manage_addFooClassForm,
            FooProduct.manage_addFooClass),
        icon='foo.gif'
    )
```

The line of primary concern to us above is the one which says permission='Add FooClasses'. This is a permission declaration which, thanks to Zope product initialization, restricts the adding of FooClasses to those users who have the *Add FooClasses* permission by way of a role association determined by the system administrator.

If you do not include a permission argument to registerClass, then Zope will create a default permission named *Add [meta-type]*. So, for example, if your object had a meta_type of *Animal*, then Zope would create a default permission, *Add Animals*. For the most part, it is much better to be explicit then to rely on Zope to take care of security details for you, so be sure to specify a permission for your object.

### 8.8.7 Designing For Security

“Security is hard.” – Jim Fulton.

When you’re under a deadline, and you “just want it to work”, dealing with security can be difficult. As a component developer, following these basic guidelines will go a long way toward avoiding problems with security integration. They also make a good debugging checklist!

- Ensure that any class that needs to work with security has *Acquisition.Implicit* or *Acquisition.Explicit* somewhere in its base class hierarchy.
- Design the interface to your objects around methods; don’t expect clients to access instance attributes directly.
- Ensure that all methods meant for use by restricted code have been protected with appropriate security assertions.
- Ensure that you called the global class initializer on all classes that need to work with security.

### 8.8.8 Using The RoleManager Base Class With Your Zope Product

After your Product is deployed, system managers and other users of your Product often must deal with security settings on instances they make from your classes.

Product classes which inherit Zope’s standard *RoleManager* base class allow instances of the class to present a security interface. This security interface allows managers and developers of a site to control an instance’s security settings via the Zope management interface.

The user interface is exposed via the *Security* management view. From this view, a system administrator may secure instances of your Product’s class by associating roles with permissions and by asserting that your object instance contains local roles. It also allows them to create user-defined roles within the Zope management framework in order
to associate these roles with the permissions of your product and with users. This user interface and its usage patterns are explained in more detail within the Zope Book’s security chapter.

If your Product’s class does not inherit from RoleManager, its methods will still retain the security assertions associated with them, but you will be unable to allow users to associate roles with the permissions you’ve defined respective to instances of your class. Your objects will also not allow local role definitions. Note that objects which inherit from many of the built-in classes such as OFS.SimpleItem.SimpleItem or OFS.ObjectManager. ObjectManager already inherit from RoleManager.

8.8.9 Conclusion

Zope security is based upon roles and permissions. Users have roles. Security policies map permissions to roles. Classes protect methods with permissions. As a developer your main job is to protect your classes by associating methods with permissions. Of course there are many other details such as protecting modules and functions, creating security user interfaces, and initializing security settings.

8.9 Testing and Debugging

| Attention:  This document was written for Zope 2. |

As you develop Zope applications you may run into problems. This chapter covers debugging and testing techniques that can help you. The Zope debugger allow you to peek inside a running process and find exactly what is going wrong. Unit testing allows you to automate the testing process to ensure that your code still works correctly as you change it. Finally, Zope provides logging facilities which allow you to emit warnings and error messages.

8.9.1 Debugging

Zope provides debugging information through a number of sources. It also allows you a couple avenues for getting information about Zope as it runs.

Product Refresh Settings

As of Zope 2.4 there is a Refresh view on all Control Panel Products. Refresh allows you to reload your product’s modules as you change them, rather than having to restart Zope to see your changes. The Refresh view provides the same debugging functionality previously provided by Shane Hathaway’s Refresh Product.

To turn on product refresh capabilities place a ‘refresh.txt’ file in your product’s directory. Then visit the Refresh view of your product in the management interface. Here you can manually reload your product’s modules with the Refresh this product button. This allows you to immediately see the effect of your changes, without restarting Zope. You can also turn on automatic refreshing which causes Zope to frequently check for changes to your modules and refresh your product when it detects that your files have changed. Since automatic refresh causes Zope to run more slowly, it is a good idea to only turn it on for a few products at a time.

Debug Mode

Normally, debug mode is set using the ‘-D’ switch when starting Zope. This mode reduces the performance of Zope a little bit. Debug model has a number of wide ranging effects:

- Tracebacks are shown on the browser when errors are raised.
• External Methods and DTMLFile objects are checked to see if they have been modified every time they are called. If modified, they are reloaded.

• Zope will not fork into the background in debug mode, instead, it will remain attached to the terminal that started it and the main logging information will be redirected to that terminal.

By using debug mode and product refresh together you will have little reason to restart Zope while developing.

The Python Debugger

Zope is integrated with the Python debugger (pdb). The Python debugger is pretty simple as command line debuggers go, and anyone familiar with other popular command line debuggers (like gdb) will feel right at home in pdb.

For an introduction to pdb see the standard pdb documentation.

There are a number of ways to debug a Zope process:

- You can shut down the Zope server and simulate a request on the command line.
- You can run a special ZEO client that debugs a running server.
- You can run Zope in debug model and enter the debugger through Zope’s terminal session.

The first method is an easy way to debug Zope if you are not running ZEO. First, you must first shut down the Zope process. It is not possible to debug Zope in this way and run it at the same time. Starting up the debugger this way will by default start Zope in single threaded mode.

For most Zope developer’s purposes, the debugger is needed to debug some sort of application level programming error. A common scenario is when developing a new product for Zope. Products extend Zope’s functionality but they also present the same kind of debugging problems that are commonly found in any programming environment. It is useful to have an existing debugging infrastructure to help you jump immediately to your new object and debug it and play with it directly in pdb. The Zope debugger lets you do this.

In reality, the “Zope” part of the Zope debugger is actually just a handy way to start up Zope with some pre-configured break points and to tell the Python debugger where in Zope you want to start debugging.

Simulating HTTP Requests

Now for an example. Remember, for this example to work, you must shut down Zope. Go to your Zope’s ‘lib/python’ directory and fire up Python and import ‘Zope’ and ‘ZPublisher’:

```
$ cd lib/python
$ python
Python 1.5.2 (#0, Apr 13 1999, 10:51:12) [MSC 32 bit (Intel)] on win32
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> import Zope, ZPublisher
```  

Here you have run the Python interpreter (which is where using the debugger takes place) and imported two modules, ‘Zope’ and ‘ZPublisher’. If Python complains about an ‘ImportError’ and not being able to find either module, then you are probably in the wrong directory, or you have not compiled Zope properly. If you get this message:

```
ZODB.POSException.StorageSystemError: Could not lock the database file. There must be another process that has opened the file.
```

This tells you that Zope is currently running. Shutdown Zope and try again.

The ‘Zope’ module is the main Zope application module. When you import ‘Zope’ it sets up Zope. ‘ZPublisher’ is the Zope ORB. See Chapter 2 for more information about ‘ZPublisher’.
You can use the ‘ZPublisher.Zope’ function to simulate an HTTP request. Pass the function a URL relative to the your root Zope object. Here is an example of how to simulate an HTTP request from the debugger:

```python
>>> ZPublisher.Zope('')
Status: 200 OK
X-Powered-By: Zope (www.zope.org), Python (www.python.org)
Content-Length: 1238
Content-Type: text/html

<HTML><HEAD><TITLE>Zope</TITLE>
... blah blah...
</BODY></HTML>

```  

If you look closely, you will see that the content returned is exactly what is returned when you call your root level object through HTTP, including all the HTTP headers.

Keep in mind that calling Zope this way does NOT involve a web server. No ports are opened. In fact, this is just an interpreter front end to the same application code the WSGI server does call.

**Interactive Debugging**

Debugging involves publishing a request up to a point where you think it’s failing, and then inspecting the state of your variables and objects. The easy part is the actual inspection, the hard part is getting your program to stop at the right point.

So, for the sake our example, let’s say that you have a ‘News’ object which is defined in a Zope Product called ‘Zope-News’, and is located in the ‘lib/python/Products/ZopeNews’ directory. The class that defines the ‘News’ instance is also called ‘News’, and is defined in the ‘News.py’ module in your product.

Therefore, from Zope’s perspective the fully qualified name of your class is ‘Products.ZopeNews.News.News’. All Zope objects have this kind of fully qualified name. For example, the ‘ZCatalog’ class can be found in ‘Products.ZCatalog.ZCatalog.ZCatalog’ (The redundancy is because the product, module, and class are all named ‘ZCatalog’).

Now let’s create an example method to debug. You want your news object to have a ‘postnews’ method, that posts news:

```python
class News(...):
    ...
    def postnews(self, news, author="Anonymous"):
        self.news = news
    def quote(self):
        return '%s said, "%s"' % (self.author, self.news)
```

You may notice that there’s something wrong with the ‘postnews’ method. The method assigns ‘news’ to an instance variable, but it does nothing with ‘author’. If the ‘quote’ method is called, it will raise an ‘AttributeError’ when it tries to look up the name ‘self.author’. Although this is a pretty obvious goof, we’ll use it to illustrate using the debugger to fix it.

Running the debugger is done in a very similar way to how you called Zope through the python interpreter before, except that you introduce one new argument to the call to ‘Zope’:
Here, you call Zope from the interpreter, just like before, but there are two differences. First, you call the ‘postnews’ method with an argument using the URL, ‘/News/postnews?new=blah’. Second, you provided a new argument to the Zope call, ‘d=1’. The ‘d’ argument, when true, causes Zope to fire up in the Python debugger, pdb. Notice how the Python prompt changed from ‘>>>' to ‘pdb>’. This indicates that you are in the debugger.

When you first fire up the debugger, Zope gives you a helpful message that tells you how to get to your object. To understand this message, it’s useful to know how you have set Zope up to be debugged. When Zope fires up in debugger mode, there are three breakpoints set for you automatically (if you don’t know what a breakpoint is, you need to read the python debugger documentation).

The first breakpoint stops the program at the point that ZPublisher (the Zope ORB) tries to publish the application module (in this case, the application module is ‘Zope’). The second breakpoint stops the program right before ZPublisher tries to traverse down the provided URL path (in this case, ‘/News/postnews’). The third breakpoint will stop the program right before ZPublisher calls the object it finds that matches the URL path (in this case, the ‘News’ object).

So, the little blurb that comes up and tells you some keys to press is telling you these things in a terse way. Hitting ‘s’ will step you into the debugger, and hitting ‘c’ will continue the execution of the program until it hits a breakpoint.

Note however that none of these breakpoints will stop the program at ‘postnews’. To stop the debugger right there, you need to tell the debugger to set a new breakpoint. Why a new breakpoint? Because Zope will stop you before it traverse your objects path, it will stop you before it calls the object, but if you want to stop it exactly at some point in your code, then you have to be explicit. Sometimes the first three breakpoints are convienent, but often you need to set your own special break point to get you exactly where you want to go.

Setting a breakpoint is easy (and see the next section for an even easier method). For example:

```
pdb> import Products
Breakpoint 5 at C:\Program Files\WebSite\lib\python\Products\ZopeNews\News.py:42
pdb>
```

First, you import ‘Products’. Since your module is a Zope product, it can be found in the ‘Products’ package. Next, you set a new breakpoint with the break debugger command (pdb allows you to use single letter commands, but you could have also used the entire word ‘break’). The breakpoint you set is ‘Products.ZopeNews.News.News.postnews’.

After setting this breakpoint, the debugger will respond that it found the method in question in a certain file, on a certain line (in this case, the fictitious line 42) and return you to the debugger.

Now, you want to get to your ‘postnews’ method so you can start debugging it. But along the way, you must first continue through the various breakpoints that Zope has set for you. Although this may seem like a bit of a burden, it’s actually quite good to get a feel for how Zope works internally by getting down the rhythm that Zope uses to publish your object. In these next examples, my comments will begin with “#”. Obviously, you won’t see these comments when you are debugging. So let’s debug:

```
pdb> s
# 's'tep into the actual debugging
> <string>(1)?()
# this is pdb's response to being stepped into, ignore it
```

(continues on next page)
pdb> c
# now, let's 'c'ontinue onto the next breakpoint

> C:\Program Files\WebSite\lib\python\ZPublisher\Publish.py(112)publish()
-> def publish(request, module_name, after_list, debug=0,

# pdb has stopped at the first breakpoint, which is the point where
# ZPublisher tries to publish the application module.

pdb> c
# continuing onto the next breakpoint you get...

> C:\Program Files\WebSite\lib\python\ZPublisher\Publish.py(101)call_object()
-> def call_object(object, args, request):

Here, ‘ZPublisher’ (which is now publishing the application) has found your object and is about to call it. Calling your
object consists of applying the arguments supplied by ‘ZPublisher’ to the object. Here, you can see how ‘ZPublisher’
is passing three arguments into this process. The first argument is ‘object’ and is the actual object you want to call.
This can be verified by printing the object:

pdb> p object
<News instance at 00AFE410>

Now you can inspect your object (with the print command) and even play with it a bit. The next argument is ‘args’.
This is a tuple of arguments that ‘ZPublisher’ will apply to your object call. The final argument is ‘request’. This is
the request object and will eventually be transformed in to the DTML usable object ‘REQUEST’. Now continue, your
breakpoint is next:

pdb> c
> C:\Program Files\WebSite\lib\python\Products\ZopeNews\News.py(42)postnews()
-> def postnews(self, N)

Now you are here, at your method. To be sure, tell the debugger to show you where you are in the code with the ‘l’
command. Now you can examine variable and perform all the debugging tasks that the Python debugger provides.
From here, with a little knowledge of the Python debugger, you should be able to do any kind of debugging task that
is needed.

Interactive Debugging Triggered From the Web

If you are running in debug mode you can set break points in your code and then jump straight to the debugger when
Zope comes across your break points. Here’s how to set a breakpoint:

import pdb; pdb.set_trace()

Now start Zope in debug mode and point your web browser at a URL that causes Zope to execute the method that
includes a breakpoint. When this code is executed, the Python debugger will come up in the terminal where you
started Zope. Also note that from your web browser it looks like Zope is frozen. Really it’s just waiting for you do
your debugging.

From the terminal you are inside the debugger as it is executing your request. Be aware that you are just debugging
one thread in Zope, and other requests may be being served by other threads. If you go to the Debugging Info screen
while in the debugger, you can see your debugging request and how long it has been open.

It is often more convenient to use this method to enter the debugger than it is to call ‘ZPublisher.Zope’ as detailed in
the last section.
Post-Mortem Debugging

Often, you need to use the debugger to chase down obscure problems in your code, but sometimes, the problem is obvious, because an exception gets raised. For example, consider the following method on your ‘News’ class:

```python
def quote(self):
    return '%s said, "%s"' % (self.Author, self.news)
```

Here, you can see that the method tries to substitute ‘self.Author’ in a string, but earlier we saw that this should really be ‘self.author’. If you tried to run this method from the command line, an exception would be raised:

```
>>> ZPublisher.Zope('/News/quote')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "./News.py", line 4, in test
    test2()
  File "./News.py", line 3, in test2
    return '%s said, "%s"' % (self.Author, self.news)
NameError: Author

>>> 
```

Using Zope’s normal debugging methods, you would typically need to start from the “beginning” and step your way down through the debugger to find this error (in this case, the error is pretty obvious, but more often than not errors can be pretty obscure!).

Post-mortem debugging allows you to jump directly to the spot in your code that raised the exception, so you do not need to go through the possibly tedious task of stepping your way through a sea of Python code. In the case of our example, you can just pass ‘ZPublisher.Zope’ call a ‘pm’ argument that is set to 1:

```
>>> ZPublisher.Zope('/News/quote', pm=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "./News.py", line 4, in test
    test2()
  File "./News.py", line 3, in test2
    return '%s said, "%s"' % (self.Author, self.news)
NameError: Author
(pdb)
```

Here, you can see that instead of taking you back to a python prompt, the post mortem debugging flag has caused you to go right into the debugging, exactly at the point in your code where the exception is raised. This can be verified with the debugger’s (l)ist command. Post mortem debugging offers you a handy way to jump right to the section of your code that is failing in some obvious way by raising an exception.

Debugging With ZEO

ZEO presents some interesting debugging abilities. ZEO lets you debug one ZEO client when other clients continue to server requests for your site. In the above examples, you have to shut down Zope to run in the debugger, but with ZEO, you can debug a production site while other clients continue to serve requests. Using ZEO is beyond the scope of this chapter. However, once you have ZEO running, you can debug a client process exactly as you debug a single-process Zope.
8.9.2 Unit Testing

Unit testing allows you to automatically test your classes to make sure they are working correctly. By using unit tests you can make sure as you develop and change your classes that you are not breaking them. Zope comes with Pyunit. You can find out more information on Pyunit at the Pyunit home page. Pyunit is also part of the Python standard library as of Python 2.1.

What Are Unit Tests

A “unit” may be defined as a piece of code with a single intended purpose. A “unit test” is defined as a piece of code which exists to codify the intended behavior of a unit and to compare its intended behavior against its actual behavior.

Unit tests are a way for developers and quality assurance engineers to quickly ascertain whether independent units of code are working as expected. Unit tests are generally written at the same time as the code they are intended to test. A unit testing framework allows a collection of unit tests to be run without human intervention, producing a minimum of output if all the tests in the collection are successful.

It’s a good idea to have a sense of the limits of unit testing. From the Extreme Programming Enthusiast website here is a list of things that unit tests are not:

- Manually operated.
- Automated screen-driver tests that simulate user input (these are “functional tests”).
- Interactive. They run “no questions asked.”
- Coupled. They run without dependencies except those native to the thing being tested.
- Complicated. Unit test code is typically straightforward procedural code that simulates an event.

Writing Unit Tests

Here are the times when you should write unit tests:

- When you write new code
- When you change and enhance existing code
- When you fix bugs

It’s much better to write tests when you’re working on code than to wait until you’re all done and then write tests.

You should write tests that exercise discrete “units” of functionality. In other words, write simple, specific tests that test one capability. A good place to start is with interfaces and classes. Classes and especially interfaces already define units of work which you may wish to test.

Since you can’t possibly write tests for every single capability and special case, you should focus on testing the riskiest parts of your code. The riskiest parts are those that would be the most disastrous if they failed. You may also want to test particularly tricky or frequently changed things.

Here’s an example test script that tests the ‘News’ class defined earlier in this chapter:

```python
import unittest
import News

class NewsTest(unittest.TestCase):
    def testPost(self):
        n=News()
```

(continues on next page)
def testQuote(self):
    n = News()
    s = 'example news'
    n.postnews(s)
    assert n.quote() == 'Anonymous said: "%s"' % s
    a = 'Author'
    n.postnews(s, a)
    assert n.quote() == '%s said: "%s"' % (a, s)

def test_suite():
    return unittest.makeSuite(NewsTest, 'news test')

def main():
    unittest.TextTestRunner().run(test_suite())

if __name__ == '__main__':
    main()
def setUp(self):
    # Get the Zope application object and store it in an
    # instance variable for use by test methods
    from Zope import app
    self.app=Zope.app()

def tearDown(self):
    # Abort the transaction and shut down the Zope database
    # connection.
    get_transaction().abort()
    self.app._p_jar.close()

    # At this point your test methods can perform tests using
    # self.app which refers to the Zope application object.

    ...

def fix_path():
    # Add Zope's lib/python directory to the Python path
    file=os.path.join(os.getcwd(), sys.argv[0])
    dir=os.path.join('lib', 'python')
    i=string.find(file, dir)
    sys.path.insert(0, file[:i+len(dir)])

def test_suite():
    return unittest.makeSuite(MyTest, 'my test')

def main():
    unittest.TextTestRunner().run(test_suite())

    if __name__=='__main__':
        fix_path()
        main()

This example shows a fairly complete Zope test fixture. If your Zope tests only needs to import Zope modules and
packages you can skip getting a Zope application object and closing the database transaction.

Some times you may run into trouble if your test assuming that there is a current Zope request. There are two ways to
deal with this. One is to use the `makerequest` utility module to create a fake request. For example:

class MyTest(unittest.TestCase):
    ...

    def setUp(self):
        import Zope
        from Testing import makerequest
        self.app=makerequest.makerequest(Zope.app())

This will create a Zope application object that is wrapped in a request. This will enable code that expects to acquire a
'REQUEST' attribute work correctly.

Another solution to testing methods that expect a request is to use the 'ZPublisher.Zope' function described earlier. Using
this approach you can simulate HTTP requests in your unit tests. For example:

import ZPublisher

(continues on next page)
If the ‘s’ argument is passed to `ZPublisher.Zope` then no output will be sent to `sys.stdout`. If you want to capture the output of the publishing request and compare it to an expected value you’ll need to do something like this:

```python
f=StringIO()
temp=sys.stdout
sys.stdout=f
ZPublisher.Zope('/myobject/mymethod')
sys.stdout=temp
assert f.getvalue() == expected_output
```

Here’s a final note on unit testing with a Zope test fixture: you may find Zope helpful. ZEO allows you to test an application while it continues to serve other users. It also speeds Zope start up time which can be a big relief if you start and stop Zope frequently while testing.

Despite all the attention we’ve paid to Zope testing fixtures, you should probably concentrate on unit tests that don’t require a Zope test fixture. If you can’t test much without Zope there is a good chance that your product would benefit from some refactoring to make it simpler and less dependent on the Zope framework.

### 8.9.3 Logging

Zope provides a framework for logging information to Zope’s application log. You can configure Zope to write the application log to a file, syslog, or other back-end.

The logging API defined in the ‘zLOG’ module. This module provides the ‘LOG’ function which takes the following required arguments:

- **subsystem** – The subsystem generating the message (e.g. “ZODB”)
- **severity** – The “severity” of the event. This may be an integer or a floating point number. Logging back ends may consider the int() of this value to be significant. For example, a back-end may consider any severity whose integer value is WARNING to be a warning.
- **summary** – A short summary of the event

These arguments to the ‘LOG’ function are optional:

- **detail** – A detailed description
- **error** – A three-element tuple consisting of an error type, value, and traceback. If provided, then a summary of the error is added to the detail.
- **reraise** – If provided with a true value, then the error given by error is reraised.

You can use the ‘LOG’ function to send warning and errors to the Zope application log.

Here’s an example of how to use the ‘LOG’ function to write debugging messages:

```python
from zLOG import LOG, DEBUG
LOG('my app', DEBUG, 'a debugging message')
```
You can use ‘LOG’ in much the same way as you would use print statements to log debugging information while Zope is running. You should remember that Zope can be configured to ignore log messages below certain levels of severity. If you are not seeing your logging messages, make sure that Zope is configured to write them to the application log.

In general the debugger is a much more powerful way to locate problems than using the logger. However, for simple debugging tasks and for issuing warnings the logger works just fine.

8.9.4 Other Testing and Debugging Facilities

There is a few other testing and debugging techniques and tools not commonly used to test Zope. In this section we’ll mention several of them.

Debug Logging

Zope provides an analysis tool for debugging log output. This output allows may give you hints as to where your application may be performing poorly, or not responding at all. For example, since writing Zope products lets your write unrestricted Python code, it’s very possibly to get yourself in a situation where you “hang” a Zope request, possibly by getting into a infinite loop.

To try and detect at which point your application hangs, use the requestprofiler.py script in the utilities directory of your Zope installation. To use this script, you must run Zope with the ‘-M’ command line option. This will turn on “detailed debug logging” that is necessary for the requestprofiler.py script to run. The requestprofiler.py script has quite a few options which you can learn about with the ‘–help’ switch.

In general debug log analysis should be a last resort. Use it when Zope is hanging and normal debugging and profiling is not helping you solve your problem.

HTTP Benchmarking

HTTP load testing is notoriously inaccurate. However, it is useful to have a sense of how many requests your server can support. Zope does not come with any HTTP load testing tools, but there are many available. Apache’s ‘ab’ program is a widely used free tool that can load your server with HTTP requests.

8.9.5 Summary

Zope provides a number of different debugging and testing facilities. The debugger allows you to interactively test your applications. Unit tests allow help you make sure that your application is develops correctly. The logger allows you to do simple debugging and issue warnings.

8.10 Appendix A: Zope Core Permissions

This is a list of standard permissions included with Zope. It is a good idea to use these permissions when applicable with your Zope products, rather than creating new ones. A list of built-in Zope permissions are available in the source code of the AccessControl package.

8.10.1 Core Permissions

• Access contents information
• Add Database Methods
• Add Documents, Images, and Files
• Add External Methods
• Add Folders
• Add MailHost objects
• Add Page Templates
• Add Python Scripts
• Add User Folders
• Add Vocabularies
• Add Z Gadfly Database Connections
• Add ZCatalogs
• Add Zope Tutorials
• Change Database Connections
• Change Database Methods
• Change External Methods
• Change Images and Files
• Change Python Scripts
• Change configuration
• Change Page Templates
• Change permissions
• Change proxy roles
• Copy or Move
• Create class instances
• Define permissions
• Delete objects
• Edit Factories
• FTP access
• Import/Export objects
• Manage Vocabularies
• Manage ZCatalog Entries
• Manage ZCatalogIndex Entries
• Manage properties
• Manage users
• Open/Close Database Connection
• Open/Close Database Connections
• Query Vocabularies
• Search ZCatalog
• Take ownership
• Test Database Connections
• Undo changes
• Use Database Methods
• Use Factories
• Use mailhost services
• View
• View management screens
• WebDAV access
• WebDAV Lock items
• WebDAV Unlock items

8.10.2 Document Template Permissions

• Change DTML Documents
• Change DTML Methods