16 Web Application Frameworks

If you don't like to do your own plumbing, you send for a plumber. If you don't like building web systems from the ground up, you use frameworks. Python's natural affinity for the web has been exploited many times to build web-system frameworks, with varying degrees of success, and the results are often freely available through the generosity of their authors. Some frameworks assume that you are working with a classic web server. Others give you more freedom by concentrating mostly on HTML generation, but even these tools often include the capability to intermingle Python and output text for more flexibility in content generation.

In this chapter, you will review the classic web server architecture, and the roles that Python can play under such a regime. Next you will go on to look at some of the possible pure Python architectures for web systems. Finally, you will meet Xitami, a lightweight web server with the capability to interact with long-running processes. Although this may not cover the whole range of server architectures available, it is a sufficiently diverse range to give you a good idea of the possibilities.

Along the way, you will learn some things that will help in many different architectural contexts. The task of interacting with web clients has many common features no matter what the eventual architecture. You also will learn an easy way to integrate databases with Python, allowing you to address all database columns as attributes of the rows that contain them. The chapter concludes with a look at HTML generation and a glance at intercomponent communications. At that stage, you will be ready to tackle the kind of problem that webs are being built to solve every day.

Information Sources

It is difficult to keep track of Python web toolkits. Cameron Laird maintains useful information on web uses of Python at `http://starbase.neosoft.com/~claird/comp.lang.python/web_python.html/`, and Paul Boddie makes a good survey of the field at `http://www.paul.boddie.net/Python/web_modules.html`. Both of these resources appear to be actively maintained at the time of writing.

Most publicly available web servers come with copious documentation. The Python code you will find comes with documentation of varying quality, ranging from "thin" to "excellent," but sadly the former tends to outweigh the latter. One of the current shortcomings of the open-source world is that more people want to write code than document it. This is a real pity, when you consider that good documentation usually increases the likelihood of software being reused by others.

The software world has always undervalued documentation (at least from the pragmatic point of view of not producing enough). The Python language is a model of its type, and although the documentation may not be perfect, Fred Drake does a very good job of ensuring that it is both useful and consistent, as anyone can see at `http://www.python.org/doc/current/`. Ka-Ping Yee presents another model site at `http://web.pydoc.org/` and simultaneously manages to demonstrate the value of his `pydoc` module. What follows is an eclectic summary of available web frameworks of various types, all of which accommodate Python somehow. The exclusion of a particular framework from this chapter does not imply a value judgment about that framework. There really are far too many Python web frameworks for a single chapter, so I have tried to choose a representative sample that gives a flavor of the current state of the art.

Web Server Architectures

If you are already familiar with how web servers are put together, feel free to skip this whole section, in which we break down the server into its component parts and discuss how these parts fit together to act as a coherent whole, meeting all the requirements for a web service. Most web servers look like Figure 16.1.

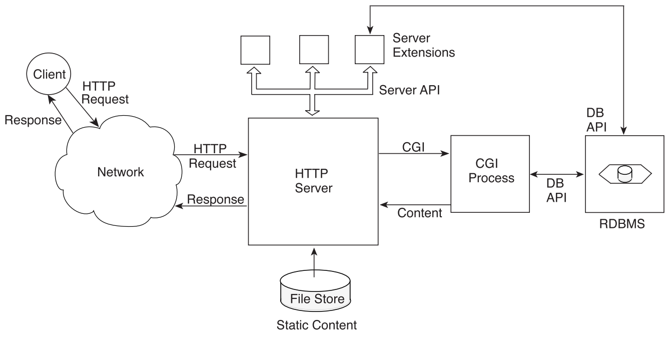


Figure 16.1 The classical web server architecture.

The process of serving *static content* is the easiest to understand. The client makes an HTTP request, the URI identifies a file in the web's virtual filestore, and the server returns the content of that file preceded by an appropriate set of HTTP response headers, including a `Content-Type` header that tells the client the MIME type of the output. Many things can go wrong along the way, but the process is normally a simple one. You are also familiar with the concept that although the client is often a browser, it might equally be any HTTP client program, possibly one of your own manufacture.

In Chapter 6, "A Server Framework Library," you received a simple introduction to the operation of the CGI mechanism, a time-honored way to activate *dynamic content*, which is to say content that is computed rather than simply retrieved and served up. The important elements are that the protocol method used by the browser can either be `GET` or `POST`. When `POST` is used, the standard input is the form content, encoded as a set of *`name=value`* pairs separated by ampersands, with the data appropriately escaped to make its interpretation unambiguous.

If a CGI routine has to use a database, then it can be a lengthy process because the following has to happen (here we consider a Python CGI, but the process is much the same with any interpreted language). First the server must locate the CGI script and verify that it is executable. Then it has to trigger the Python interpreter to read and execute the script (here Python at least precompiles modules as it imports them, although the main program would require precompilation—for example, by the compile program in Python's tools directory—to be run from its compiled form). The interpreter will then run the program, using standard input and/or the query string and/or the HTTP headers for inputs, and reading and writing files or relational databases as necessary.

This means that each CGI script carries the overhead of creating a new process, starting up the Python interpreter, and creating any required database connections or opening required files before it can start to process the inputs from the client. No wonder that various solutions exist for the elimination of the CGI script!

The real issue is therefore application scalability: if dynamic content can be created with lower overhead, then more web pages can be produced in a given period of time. If your server is lightly loaded, process startup overhead might be a complete non-problem. As with network bandwidth, the situation deteriorates rapidly when the demand outstrips the supply and the server becomes a saturated resource. Just the same, it is comforting to remember that CGI scripting is compatible with almost all web servers and is easily achieved in Python.

One of the most popular early ways of omitting the overhead was for a server to define an extension API. Server extensions use the API to use server functionality and are loaded into memory as the server starts. When a request is associated with a server extension, the server can simply call the extension code directly, with no need to create a separate process. Of course, the server may or may not be threaded: if it is, then several threaded copies of it may be executing concurrently on separate requests, and if extensions are written in a thread-safe way, it is much easier for the server to scale up by adding threads as it gets busier. Your choices here depend on the relative costs of process and thread creation in the operating system platform that supports your server.

Another advantage of the server extension is that if database connections are required, they can be created on demand (or when the server starts) and pooled among the various pages that use them. Effectively, the extension can implement persistent storage on behalf of any web application because it is a part of a permanent process, which continues to exist between requests. For this reason, server extensions such as ColdFusion have been very popular, not only offering a useful increment in performance over traditional CGIs but also automatically implementing easy-to-use state-maintenance mechanisms.

Different servers have defined different extension APIs, and one of the problems of providing functionality as server extensions is the nonportable nature of the solution. The technique has therefore been much more popular with the more widely used servers such as Apache and Internet Information Server. Although it comprises less than 1 percent of the Internet web server population, the open-source AOLserver has a loyal following among sites generating a lot of traffic, and it too has received attention from the Python community.

CGI remains the baseline (or lowest common denominator, depending on your point of view). It is almost universally available, with only the Medusa server standing out as one that eschews the CGI function. It does so primarily because the process creation required by CGI conflict's with Medusa's design goals, which favor a small footprint and fast response.

Apache Integration with Python

The Apache server is a development of the earlier NCSA server, and is an open-source product, just approaching a major second release. Apache is the favored web server in the UNIX environment, and it continues in its long-time position as the most popular web server on the Internet (you can check the usage statistics at `http://www.netcraft.com/survey/`). It provides a module API, which has been used to implement many server extensions. Apache processes each request in a number of phases, and each phase may call a number of handler functions. Handler functions from optional modules can be invoked at certain points in each phase if they are correctly installed and described in the Apache configuration file.

Apache was originally implemented under UNIX, but it is now also available for Windows NT and 2000, with an experimental version available for Windows 9X. This latter version appears to be relatively stable and supports standard Python CGI scripts quite well, albeit slowly because of the overhead of starting up the Python interpreter for each request. Apache is therefore a good choice if you might want to migrate from one OS platform to another. It also supports a number of Python-oriented extensions of varying levels of sophistication. The three best known, in order of increasing complexity, are PyApache, `mod_python`, and `mod_snake`. All are open source, giving you complete freedom to change what you do not like.

PyApache

PyApache by Lele Gaifax has a well-run resource center web site (`http://bel-epa.com/pyapache/`) maintained by Graham Higgins. PyApache embeds the Python interpreter into Apache as an extension module. Effectively, the server loads the Python interpreter when it starts and then initializes it for each script it must execute, making the language readily available for CGI scripting without the interpreter having to be loaded for each script. PyApache has reasonably good documentation about how to use it, with excellent advice on modifying your web server configuration file and some lively CGI examples surrounded by common-sense advice on defensive programming techniques in a web environment.

Although using PyApache overcomes the problem of process creation for CGI scripts, clearly each script is run as a standalone object, and therefore must create its own database connections and so on, which will be released when the script ends. The nice part about PyApache is that absolutely any standard CGI script will work with it. You can easily port webs that already work using Python CGI to the PyApache environment, or simply install PyApache in your existing Apache server and thereby gain a useful performance increment.

mod_python

This module, maintained by Gregory Trubetskoy at `http://www.modpython.org`, is much broader in initial concept than PyApache, allowing you to define your own handler module functions in Python. You effectively nominate a Python module as a handler for certain classes of URL. The handler module is registered in the Apache configuration file using a structure like the following:

```
<Directory /pscripts>
    AddHandler python-program .py
    PythonHandler myscript
</Directory>
```

The argument to the `<Directory>` directive is a physical directory, and this configuration specifies that any request for a `*.py` file from the `/pscripts` directory should be handled by the script `/pscripts/myscript.py`. In this particular case, `PythonHandler` is the generic handler, which handles the phase during which content is generated (the authentication handler would be `PythonAuthenHandler`, for example). When `mod_python` arrives at the generic handler phase, it does the following:

1. Inserts the directory to which the Python Handler directive applies at the start of `sys.path` (unless this has already been done).
2. Attempts to import the handler module (in this case, `myscript`) by name.
3. Calls the script's `handle()` function, passing it a `Request` object (which encodes details of the current request and which the handler can use to set response attributes such as the MIME type of the returned content).

Here is a suitable handler, which would be located at `/pscripts/myscript.py`:

```
from mod_python import apache

def handler(request):
    req.content_type = "text/plain"     # establish content type
    req.send_http_header()              # send headers to client
    req.write("Hello, Python users!")   # send output
    return apache.OK                    # inform server no errors
```

The particular handler directive shown previously, which installed this handler, specifies that the script be called for *any* URL ending in `.py`. The handler you see simply ignores the URL and sends the same output. A more adventurous handler might return `Content-Type: text/html` and send back an HTML version of the Python code indicated by the URL path with appropriate syntax coloring. Another handler might simply treat the Python script named in the URL as a CGI script and attempt to execute it.

Clearly, `mod_python` is more intimately integrated with Apache than PyApache is, and this gives it more flexibility—at the usual cost of additional complexity, though if you are running Apache and you program in Python there is nothing to fear. `mod_python` does not integrate fully with the server, however. It does not allow the use of embedded Python in content, for example. Also, at the time of writing, `mod_python` was only available for the Apache 1.3 implementation. There is no indication of how likely it is to be available for the forthcoming Apache 2.0 release, in which the internals have changed somewhat.

mod_snake

This interesting extension, maintained by Jon Travis at `http://modsnake.sourceforge.net/`, is designed to interoperate with both the Apache 1.3 and 2.0 architectures. It integrates with Apache to give Python modules the capability to do anything that a module coded in the C language could do. Because the majority of users want to create simple modules, accelerate CGI scripts, or embed Python in their web pages, `mod_snake` also provides specialized (and simpler) APIs to perform these common tasks.

CGI script acceleration is provided by `mod_snake_cgi`, which requires the following line in the `httpd.conf` Apache configuration file:

```
SnakeModule     mod_snake_cgi.SnakeCGI
```

Notice that in `mod_snake`, you can define new modules as well as new handlers. You also have to associate particular paths with the module to ensure that scripts in certain areas of your web are run as Python CGIs. A typical configuration entry for this purpose follows:

```
Alias /pythoncgi/ ''/home/httpd/apache/cgipy/''
<Directory ''/home/httpd/apache/cgipy''>
    SetHandler snakecgi
    AllowOverride None
    Options ExecCGI
    Order allow,deny
    Allow from all
</Directory>
```

When Apache now sees a request of the form `http://your.host.com/pythoncgi/foo.py`, it will run it via `mod_snake_cgi`, looking for the script in the aliased directory `/home/httpd/apache/cgipy`. Before running a Python CGI script, `mod_snake_cgi` checks an internal cache of already-loaded modules. If the script is not yet loaded, then it is imported and added to the list of loaded modules. Otherwise, its modification time is checked, and if this is later than the last load, the Python module is reloaded to ensure that the new version will be run. The compiled code then runs in its entirety.

To allow embedded Python in your web pages, in a similar (but more advanced) way to ASP processing under IIS, you would need to add the following directives to your main Apache configuration file:

```
SnakeModule mod_snake_epy.SnakeEPy
AddType application/x-httpd-epy .epy
```

This tells Apache that scripts whose names end in `.epy` should be processed as extended Python scripts. There are configuration options you can use to

- Set processing of embedded Python code on or off.
- Send errors to the client instead of the log file (useful when you are debugging a new set of scripts because you no longer have to continually review the error file).
- Log extended Python errors somewhere other than the standard error log file.

Within an extended Python script, there are three delimiters you can use to indicate the inclusion of Python code. They are distinguished by different escape sequences for the three different purposes to which code might be put. Table 16.1 lists these delimiters.

Table 16.1 ***mod_snake*** **Python Markup and Its Interpretation**

Delimiters	Purpose of Code
`<+ ... +>`	Initialization code that should run only the first time the script executes. This can be used to open database connections, read in data files, and so on.
`<¦ ... ¦>`	To be executed with return value discarded.
`<: ... :>`	To be executed with the return value replacing the bracketed expression in the content sent to the client.

The CGI code modules have a global object `EPY` made available in their namespace before execution, which has several useful methods. A CGI script can use `EPY` to

- Set a script to be persistent, in which case global objects created during execution will be available during subsequent requests.
- Set output headers to be returned to the client.
- Write data to the remote client.

Here is a simple example from the `mod_snake` documentation to show the code in use:

```
<+
import time

EPY.set_persist(1)
HitCount = 1
+>

The current time is: <:time.ctime(time.time()):><BR>
This page has been hit <:HitCount:> times.
<¦HitCount = HitCount + 1¦>
```

This module sets itself persistent the first time it is run, so the hit count is only initialized once, and the variable will be available to successive invocations of this script (as will module time). The content returned to the client will contain interpolated output of the two Python expressions, which give the time and the hit count, respectively. Finally, the hit count is incremented, but nothing further is inserted into the output stream.

mod_fastcgi

FastCGI is a general mechanism described at `http://www.fastcgi.com/` and intended to be applicable without limitations as to language or web server architecture. FastCGI processes persist across requests, solving the interpreter startup problem. By multiplexing standard input, standard output, and standard error over a single pipe or network connection, it allows web processes to run remotely from the server.

FastCGI modules can be any of the following:

- *Responders*, which play the traditional role of CGI functionality.
- *Filters*, which apply computation to a resource identified by the web server and can achieve performance improvements by caching, for example.
- *Authorizers*, which receive the same information as a CI script would, but are only required to return an HTTP response indicating whether the request is acceptable.

Robin Dunn provides one implementation of FastCGI for Python, tested with both the Apache and Stronghold web servers, at `http://alldunn.com/python/`. FastCGI can speed up CGI performance significantly without unduly complicating your scripts and is worth a look if other techniques are unavailable or do not yield good enough performance.

AOLserver Integration with Python

AOLserver is maintained on SourceForge (at `http://sourceforge.net/projects/aolserver/`) by Kris Rehberg and Jim Davidson with the help of many other AOL staff members. It is an open-source, enterprise-level web server, which differs from Apache in some fundamental ways. It is a multithreaded application, whereas Apache runs client processes to handle requests (although multithreading is to be added in Apache's 2.0 release). This does not imply deficiencies in the Apache architecture, which by choice prefers flexibility to efficiency.

AOLserver also has its own database-independent API, which allows the server to establish a pool of database connections and then share them among the various worker threads as necessary—an ODBC interface has recently been added. Finally, AOLserver includes a complete implementation of the Tcl language along with many Tcl routines designed to assist with common tasks, such as retrieving connection information and handling forms.

Python access to many of AOLserver's functions is available through the PyWX module maintained by C. Titus Brown, Brent A. Fulgham, and Michael R. Haggerty at `http://pywx.idyll.org/`. This module's major attractions are

- Support for threaded execution, with the option to reuse the same interpreter on multiple threads or invoke a separate one.
- Access to AOLserver's internal C API, as well as the Tcl language implementation.

- Persistent database connections via the AOLserver internal implementation.
- An optional CGI-compatible environment with compiled script caching similar to `mod_snake`.

The configuration information available is complete, but it assumes a reasonable familiarity with the structure of AOLserver, so there is little point repeating it here. The AOLserver/PyWX combination seems to be powerful, but at the same time some of the demonstration scripts available on the web site note that "if you get strange results, it could be that someone else is running the scripts at the same time." One might hope that this is because the scripts are deliberately simplified.

Internet Information Server Integration with Python

Microsoft's IIS is widely used on intranets, and on the Internet to a lesser degree, by organizations whose standard server platform is Windows NT or Windows 2000. This includes some major web hosting corporations. IIS can use Active Scripting, and the usual solution is to run VBScript pages with an `.asp` extension, thereby invoking the VBScript interpreter for blocks enclosed in `<% ... %>` delimiter pairs. Microsoft also supplies Personal Web Server (PWS), which offers an ASP scripting environment sufficiently close to IIS for testing purposes.

Active Scripting, however, is not limited to just VBScript. By default, the engine will also allow you to interpret server-side JavaScript (or Microsoft's implementation of it) for scripts characterized as follows:

```
<SCRIPT LANGUAGE="JScript" RUNAT="Server">
```

If you install the Python win32all extensions, Python also is installed as an Active Scripting language, and you can use `LANGUAGE="Python"` to trigger it. However, all is not sweetness and light, for a number of reasons:

- Indentation is a big problem if you try to intermingle HTML and program code as is common in VBScript. The best workaround for this problem is "don't do that!"—use programming to create HTML from Python as discussed later in this chapter, and output it only when you have generated it all.
- Do not try to be too literal in your translations because VBScript syntax takes a number of liberties (such as allowing function calls with no parentheses or arguments), which Python does not.
- You need some insight into the ASP object model to know exactly how to refer to each object type. Many ASP objects make information available as *collections*, which are ASP's equivalent of the Python dictionary. However, a collection can have a default item, which the collection returns when its name is not qualified by an attribute name.

You also need to be careful to ensure that your Python pages are named with the `.asp` extension; otherwise, they will not be treated as Active Server Pages at all, much to your chagrin. This probably trips up 50 percent of those who start down the Python/ASP road (including your author).

One of the features of Active Server Pages is the object hierarchy it makes available to you. This includes session and application objects, which if carefully used allow automatic maintenance of data storage between successive interactions with a particular browser, or for all browsers accessing a subset of the virtual web defined at the server as an application. This feature is useful, but storing randomly created COM objects in session or application space is not a good idea. After initially promoting the session object as a suitable way to maintain state, Microsoft later backpedaled when users began storing large record sets in the session to save repetitious database access.

The ASP environment automatically makes available several useful objects:

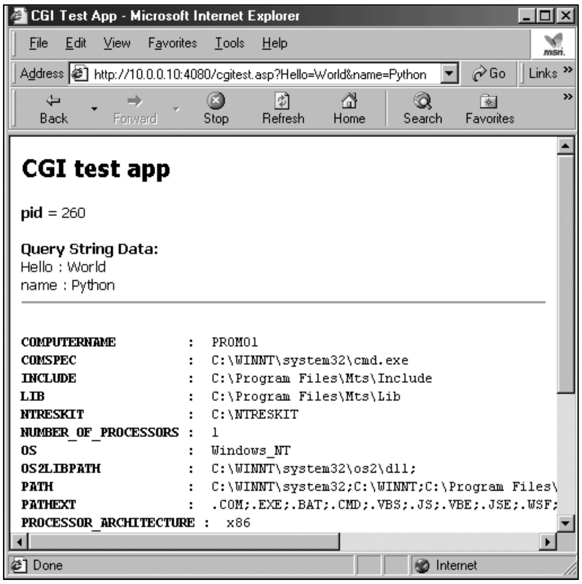
- **`Application`**—ASP defines an application as a directory in the web's virtual filestore, plus all its subdirectories that are not themselves applications. You can use the application object to store data that all pages need to share, avoiding the overhead of creating them on a page-by-page basis. Offers `Lock()` and `Unlock()` methods to avoid synchronization problems on multisession access.
- **`ASPError`**—Various properties of this object can be read to determine information about an error.
- **`ObjectContext`**—Used to control interactions with Microsoft Transaction Server.
- **`Request`**—Used to access the data that characterize the client's request. Properties include `Certificates`, `Cookies`, `Form`, `QueryString`, and `ServerVariables`.
- **`Response`**—By far the most complex of these objects, `Response` has many attributes and methods, of which the most important are `Cookies` (an attribute that returns cookies to the client), `Write` (used to send output to the client), and `AddHeader` (which adds an HTTP header to the response seen by the client).
- **`Server`**—The `CreateObject()` method can be used to create an arbitrary COM object. Python can use COM objects almost as easily as native Python objects, thanks to Mark Hammond's win32 extensions. The `HTMLEncode()` and `URLEncode()` methods offer features that would otherwise require additional Python modules to be loaded, and the `GetLastError()` method returns an `ASPError` object detailing the last ASP error that occurred.
- **`Session`**—The major use of the `Session` object is for storage of data that need to persist at the server between successive client interactions. It also has a number of useful methods, including `Abandon()` to terminate the session unilaterally and `Contents.Remove()` to remove a named item from session storage.

The `Request` object will actually search a number of collections for an attribute, so if you know the attributes you want, they will be retrieved from *either* the `Form` or the `QueryString` collection as necessary. Some software authors use this technique to produce ASP scripts that will work with either `GET` or `POST` methods.

Listing 16.1 shows a simple ASP page, which shows how you can use the `QueryString` attribute of the `Request` object. `QueryString` is a collection of the names and values from the URL's query string. As you can see from line 12, Python sees the default value as a list of the keys. In line 13, dictionary access is used to retrieve the values of individual values. Figure 16.2 shows the output of this page when run on my IIS server. The `Form` object, filled from the script's standard input when first accessed, provides similar access for method `POST` calls. The method of HTML production is typical of Python ASP programs because it avoids the need to intermingle Python and HTML.

Listing 16.1 ***cgitest.asp:* A Simple CGI Test Script from the ASP Environment**

```
 1  <%@ LANGUAGE="PYTHON" %>
 2  <%
 3  def _test():
 4      import os, sys, os, string
 5
 6      eol = "\r\n"
 7      doc = ['<HTML><HEAD><TITLE>CGI Test App</TITLE></HEAD>\r\n<BODY>']
 8      doc.append('<H2>CGI test app</H2><P>')
 9      if hasattr(os, 'getpid'):
10          doc.append('<b>pid</b> = %s<br>' % os.getpid())
11      doc.append('<br><b>Query String Data:</b><BR>')
12      for ff in Request.QueryString:
13          doc.append(ff + " : " + Request.QueryString[ff] + "<BR>")
14      doc.append('<HR><P><pre>')
15      keys = os.environ.keys()
16      keys.sort()
17      for k in keys:
18          doc.append('<b>%-20s :</b>  %s' % (k, os.environ[k]))
19      doc.append('\n</pre><HR>')
20      doc.append('</BODY></HTML>')
21
22
23      Response.Write(string.join(doc, '\r\n'))
24
25
26  _test()
27  %>
```


CGI Test App - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://10.0.0.10:4080/cgitest.asp?Hello=World&name=Python
Go
Links
Back
Forward
Stop
Refresh
Home
Search
Favorites
Done
Internet

CGI test app

pid = 260

Query String Data:
Hello : World
name : Python

```
COMPUTERNAME          :  PROMO1
COMSPEC               :  C:\WINNT\system32\cmd.exe
INCLUDE               :  C:\Program Files\Mts\Include
LIB                   :  C:\Program Files\Mts\Lib
NTRESKIT              :  C:\NTRESKIT
NUMBER_OF_PROCESSORS  :  1
OS                    :  Windows_NT
OS2LIBPATH            :  C:\WINNT\system32\os2\dll;
PATH                  :  C:\WINNT\system32;C:\WINNT;C:\Program Files\
PATHEXT               :  .COM;.EXE;.BAT;.CMD;.VBS;.JS;.VBE;.JSE;.WSF;
PROCESSOR_ARCHITECTURE :  x86
```

Figure 16.2 Output from `cgitest.asp` running on IIS 4.0 on Windows NT.

The COM objects you can create using `Server.CreateObject()` include ActiveX Data Objects (ADO), now Microsoft's preferred way to access ODBC- and OLEDB-accessible databases. This is a useful way of accessing the ADO model, which some people find more natural than the Python DB API. Because it was designed to integrate into Microsoft operating systems, the ADO model offers features that cannot be made available via ODBC, for example, but that come "out of the box" when using the more advanced OLE drivers.

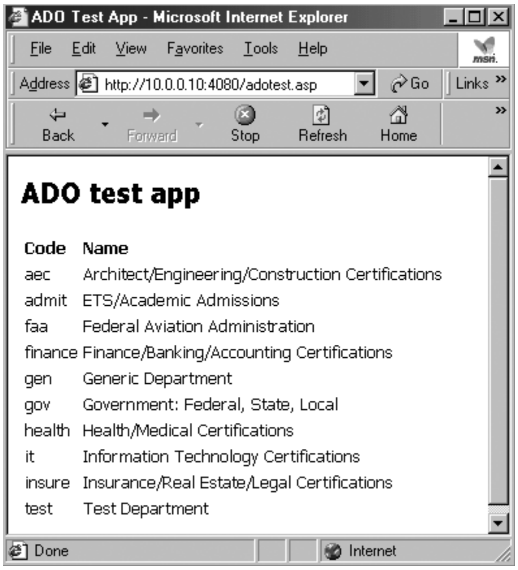
The three main objects in the ADO object model are `Connection`, `RecordSet`, and `Command`. These can combine together in a confusing number of ways, so most programmers tend to find the way that works most easily for them and stick with that. RecordSets are interesting objects because they can be detached from their data source, modified, and then later reattached and used to update the database. Listing 16.2 shows a simple use of the ADO primitives to retrieve two columns from a table.

Listing 16.2 ***adotest.asp:* Using ADO Objects from Python**

```
 1  <%@ LANGUAGE="PYTHON" %>
 2  <%
 3  def _test():
 4      import os, sys, os, string
 5
 6      eol = "\r\n"
 7      doc = ['<HTML><HEAD><TITLE>ADO Test App</TITLE></HEAD>\r\n<BODY>']
 8      doc.append('<H2>ADO test app</H2><P>')
 9      oConn = Server.CreateObject("ADODB.Connection")
10      oConn.open("prom20", "promuser", "promuser")
11      rs = Server.CreateObject("ADODB.Recordset")
12      rs.Open("SELECT DptCode, DptName FROM department ORDER BY DptName", oConn)
13      rs.MoveFirst()
14      doc.append("<TABLE>")
15      doc.append("<TR><TD><B>Code</B></TD><TD><B>Name</B></TD></TR>")
16      while not rs.EOF:
17          doc.append("<TR><TD>%s</TD><TD>%s</TD></TR>" %
18              (rs.Fields.Item("DptCode"), rs.Fields.Item(1).Value))
19          rs.MoveNext()
20      doc.append('</TABLE>')
21      doc.append('</BODY></HTML>')
22      rs.Close()
23      oConn.Close()
24
25      Response.Write(string.join(doc, eol))
26
27  _test()
28  %>
```

Note that on line 18, two alternative ways are shown to access a field's value. The first uses the field's name as an argument to a call of the record set's `Item` attribute and relies on the fact that `Value` is the default attribute. The second uses an explicit numeric argument to specify its position and explicitly requests the `Value` attribute. The output is shown in a browser window in Figure 16.3.

As I write, Microsoft is enthusiastically promoting its .NET framework, which includes an ADO.NET enhancement to its Active Data Objects technology. It appears that even Microsoft felt that there were too many ways to achieve the same end in ADO because it has rationalized the object model somewhat and removed some of the redundancy. The Visual Basic.NET language, an update to Visual Basic with many backward incompatibilities, removes the idea of default attributes for objects.


ADO Test App - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://10.0.0.10:4080/adotest.asp
Go
Links
Back
Forward
Stop
Refresh
Home
Done
Internet

ADO test app

```
Code   Name
aec    Architect/Engineering/Construction Certifications
admit  ETS/Academic Admissions
faa    Federal Aviation Administration
finance Finance/Banking/Accounting Certifications
gen    Generic Department
gov    Government: Federal, State, Local
health Health/Medical Certifications
it     Information Technology Certifications
insure Insurance/Real Estate/Legal Certifications
test   Test Department
```

Figure 16.3 Output from `adotest.asp` running on the author's NT system.

Xitami Integration with Python

Xitami (`http://www.xitami.com`) is an open-source server, intriguing to those interested in lightweight web processing. Not only does it support web operations, it also includes an FTP server so that you can easily manage your web content remotely. Its design philosophy is to keep the server as small and fast as possible, and to give it as many interesting ways as possible to interact with other components. Xitami is based on SMT multithreading software from iMatix Corporation and has been ported to an amazing number of platforms. As well as supporting UNIX of various flavors, Xitami also runs on Windows NT, 2000, and 9X as well as OS/2 and OpenVMS.

Xitami is an easily maintained product; most operator actions can be performed through the administrative web pages, which are unusually well composed. It handles virtual hosting to serve several web sites from the same IP address and port number, it supports persistent CGI, you can write your own modules using iMatix's libraries, and it also supports the *long-running web process* (LRWP), which gives the server a huge increase in versatility and performance.

Xitami Architecture and Long-Running Web Processes

The basic architecture appears in Figure 16.4. It is quite similar to the standard web server, except that as well as a standard extension API (the ISAPI interface offered by Microsoft's IIS, only available in Xitami on Windows platforms) there is a socket interface to LRWPs. These processes connect and register to serve an application by name with the server. Xitami associates application names with particular paths. If the application name and the path differ, you have to configure the paths in Xitami's configuration file.

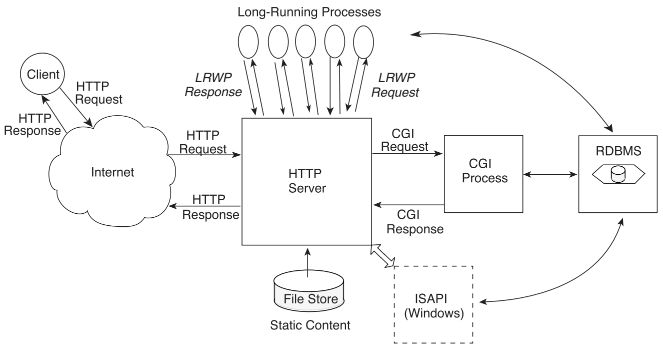


Figure 16.4 Xitami architecture showing long-running web processes.

The parallel FTP server not shown in Figure 16.4 is handy for updating web content, for example, and downloading statistics (yes, Xitami does keep statistics, in several standard and therefore easily analyzed formats).

When the server receives a request for a path matching a registered application, if one or more registered processes is ready, then the request is encoded and passed across the socket connection to a free LRWP serving that application. The process decodes the transmission, computes its result, and sends it back across the socket interface to the Xitami server, which relays it back to the web client. The LRWP then waits for its next request.

The beauty of this scheme is its scalability. Each LRWP can create its own database connections and perform any necessary initialization before it even connects to Xitami to register its availability. Because Xitami is multithreaded, if several LRWPs are registered for the same application, Xitami sends out requests to as many concurrent LRWPs as are currently registered, allowing parallelism at the process level with no per-call startup penalty. Requests that Xitami cannot handle immediately (because all LRWP server processes are busy) sit in a queue, and Xitami passes them out to LRWPs as they become available.

In the absence of a registered process for an application, Xitami simply treats requests as normal and looks for a document in the web filestore in the traditional way. This is useful if you only want to process a given number of requests. If the processing LRWP terminates when it has processed the required number of requests, further clients simply see an HTML or CGI script announcement (whose path is the same as the application's) that the server will accept no further requests. It is also possible to start LRWPs automatically under the control of a CGI script.

The CGI interface is standard, and you should expect a script that runs under Apache to run under Xitami without modification. The core Python `cgi` library module allows you to process the query string or the form contents as necessary. Xitami uses the first line of the CGI script to determine which processor will process it, so on Windows 98, my Xitami CGI scripts all begin with the following line, although you can probably understand that the path can be omitted on a correctly configured server:

```
#! D:/Python20/python
```

You may be wondering about the LRWP interface. This is the wonderful part: a Python LRWP interface module is available as a part of Xitami, and it makes the task of registering a Python process with Xitami and accepting requests very straightforward. The whole module contains fewer than 300 lines of code (including test code), and after it has registered with Xitami, the LRWP requests look like CGI requests as far as the LRWP process is concerned. It is easy to turn a standard CGI script into an LRWP.

Experiments with Xitami

Xitami is an extremely interesting model for lightweight webs built of cooperating processes. A major example in this chapter is based on Xitami's LRWP model as a typical web speedup technique. To get the most from the example, you would be well-advised to download Xitami and run the code with it. The download is a relatively small one, and Xitami's capability to run on Windows 9X as well as many other platforms makes this a worthwhile exercise for you.

You might find that the `lrwplib.py` module needs editing. In the version I downloaded, line 104 reads

```
    self.sock.connect(self.host, self.port)
```

when it should have read

```
    self.sock.connect((self.host, self.port))
```

This minor edit reflects a common usage bug in socket connections, where the host and port are separate arguments rather than the elements of a tuple. The socket libraries only stopped accepting these erroneous calls after release 1.5.2. The `lrwplib` module is exceptionally clear code. You should have no trouble understanding it, and I encourage you to read it.

Each copy of Xitami (and many may run on a single host) is configured with a "base port" number. If this is zero, then FTP service appears on port 21, web service on port 80, and LRWP processes connect to port 81. The base port number is added to each of these port numbers, so it is the easy way to adjust your port numbers to avoid conflict with any services you may already have running. You can also configure Xitami to run several virtual hosts and distinguish between requests to the same IP address by the symbolic (DNS) hostname to which they were directed. Finally, different Xitami servers can be attached to different IP addresses on the same machine.

Listing 16.3 shows a simple LRWP program, using the standard `lrwplib` module to communicate with the local Xitami server whose port base is zero. The URL to access this process will be something like

```
http://hostname/lrtest/
```

If you have chosen some other port base, then you will need to adjust the port number on line 16 and modify the URL to access port 80+ (Xitami base port).

Using Python LRWP Programs with Xitami

When running with a zero port base to give HTTP service on port 80, the Xitami server makes non-standard use of port 81 to listen for connecting long-running web processes. This normally will not matter because the service associated with port 81 is rarely used.

For your LRWP services to be available through Xitami, you must take the following steps:

1. Configure and run the Xitami server.
2. Run one or more copies of the LRWP code, checking that each one successfully attaches to the Xitami server.
3. Access the correct service URL with a suitable web client.

Because they connect to a network socket, the LRWP programs do not need to be installed in any particular directory.

Listing 16.3 ***lrwpskel.py:*** **A Simple Xitami Long-Running Web Process**

```
 1  """A small long-running web process for Xitami.
 2
 3  Takes a run-time argument and returns this plus a call count
 4  each time it is called, terminating after ten calls."""
 5
 6  import sys
 7  myname = sys.argv[1]
 8  callcount = 0
 9
10  import lrwplib
```

```
#
# One-time LRWP startup logic: connect to local Xitami
# to serve a fixed number of application "lrtest" requests
#
try:
    lrwp = lrwplib.LRWP("lrtest", '127.0.0.1', 81, '')
    lrwp.connect()
    print "Connected to Xitami"
    sys.stdout.flush()
except:
    raise # comment this out when connection works
    sys.exit("Could not start long-running web process.")

while 1:
    #
    # Per-request code
    #
    request = lrwp.acceptRequest()      # blocks until server has work
    query = request.getFieldStorage()   # retrieve task as a CGI call
    callcount += 1
    #
    # Page generation logic
    #
    #request.out.write("""Content-Type: text/html\n\n""")
    request.out.write("""<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2➥
    Final//EN">
    <html>
    <head>
    <title>LONG-RUNNING WEB PROCESS %s</title>
    </head>
    <body>
    """ % myname)
    request.out.write("""<H1>Process %s, call %d</H1>""" % (myname,➥
    callcount))
    request.out.write("""
</body>
</html>
""")
    request.finish()
    if callcount == 10:
        break

#
# LRWP Process termination
#
lrwp.close()
```

Connection to the Xitami server is extremely straightforward: create an `lrwplib.LRWP` object, passing it the name (web path) of the application you want to serve and the IP address and port of the server you want to connect to. You should clearly understand that *a Xitami LRWP need not run on the same host as the Xitami server it serves*. This gives additional scalability by allowing application servers to be LAN hosts whose processes do not compete for resources with the web server.

The remainder of the program is a simple loop in which the process waits for a request from Xitami, puts it into a `FieldStorage` object (just like a regular CGI call would do), and produces an HTML response, which in this case is very simple. The server process and its run number are identified to make it easy for a client or clients to know which process is responding.

Initial experiments showed that it was difficult to drive a set of browsers with the mouse in a way that would give meaningful loads on the server and allow you to see the action of Xitami distributing work among a number of LRWPs. You probably will find the client program shown in Listing 16.4 useful for testing: it is a modification of Chapter 6's asynchronous web client. `lrwpcli.py` makes 10 concurrent calls to the same URL and only prints lines coming back from the server that contain `<H1>` HTML tags.

Listing 16.4 ***lrwpcli.py:* An Asynchronous Client for *lrwpskel.py***

```
import syncope
import socket

class http_client (asyncore.dispatcher):

    def __init__ (self, host, path, cnum):
        asyncore.dispatcher.__init__ (self)
        self.path = path
        self.cnum = cnum
        self.host = host
        self.wflag = 1
        self.create_socket (socket.AF_INET, socket.SOCK_STREAM)
        self.connect ((self.host, 80))

    def handle_connect (self):
        self.send ('GET %s HTTP/1.0\r\n\r\n' % self.path)
        self.wflag = 0

    def handle_read (self):
        data = self.recv (8192)
        if not data:
            return ""
        lines = data.split("\n")
        for line in lines:
            if line.find("<H1>") >= 0:
```

```
26                  print "Channel:", self.cnum, ">>>", line
27
28      def handle_close(self):
29          self.close()
30
31      def handle_write (self):
32          return
33
34      def writable(self):
35          return self.wflag
36
37  import sys
38  import urlparse
39  cnum = 0
40  for url in ["http://127.0.0.1/lrtest/"]*10: # or as you please ...
41      parts = urlparse.urlparse (url)
42      if parts[0] != 'http':
43          raise ValueError, "HTTP URL's only, please"
44      else:
45          cnum += 1
46          host = parts[1]
47          path = parts[2]
48          http_client (host, path, cnum)
49  asyncore.loop()
```

You have seen a similar program earlier in the book, when you were studying asynchronous server processes. Line 40 begins a loop that submits 10 requests from the web server and processes the responses from them.

You can vary this script easily for your own testing purposes. Listing 16.5 shows the output from the program when three `lrwpskel.py` LRWP processes were running.

Listing 16.5 **Output from a Run of *lrwpcli.py* with Three Active *lrwpskel* Processes**

```
D:\Book1\Code\Ch16>python ashttpcli.py
Channel: 2 >>>     <H1>Process ONE, call 5</H1>
Channel: 5 >>>     <H1>Process TWO, call 4</H1>
Channel: 6 >>>     <H1>Process THREE, call 4</H1>
Channel: 8 >>>     <H1>Process ONE, call 6</H1>
Channel: 4 >>>     <H1>Process TWO, call 5</H1>
Channel: 10 >>>     <H1>Process THREE, call 5</H1>
Channel: 3 >>>     <H1>Process ONE, call 7</H1>
Channel: 7 >>>     <H1>Process TWO, call 6</H1>
Channel: 9 >>>     <H1>Process THREE, call 6</H1>
Channel: 1 >>>     <H1>Process ONE, call 8</H1>
```

Significance of Xitami and Long-Running Web Processes

A Xitami server process accepts registrations from long-running web processes on port 81. Each LRWP registration provides the web path it will serve; this is the first argument to the `lrwplib.LRWP()` call, which parameterizes the connection between the LRWP and the Xitami server. When Xitami receives a request from a web client for a web path served by an LRWP, it passes the request data out to the appropriate LRWP. If the LRWP is busy, then Xitami holds the request in a queue until it can be served.

There is nothing, however, to stop multiple LRWPs registering *to serve the same web path*. In this case, the Xitami server treats the LRWPs serving the same path as a pool: when a request arrives for the path, it need not be queued in Xitami if there is an inactive LRWP available to serve it. Each web path served by LRWPs is therefore scalable, simply by running multiple copies of the LRWP and letting Xitami act as a queuing request multiplexer.

You can think of a group of LRWPs serving the same application as an application group. Each application can be served by as many LRWPs as are required for adequate performance. Different servers can serve the same application group if one or more LRWPs register for the application with each web server. This is not the limit of Xitami's scalability, however. One host can run several instances of Xitami, each using a different base port number. Because the architecture is lean and multithreaded, this does not impose a heavy load on the host processor under quiescent conditions. Each Xitami server can accept registrations from LRWP application servers, so the same application can be served at many different HTTP ports, each by a Xitami server that is time-sharing with the other Xitami servers and the LRWPs.

A host running several copies of Xitami might be represented by the diagram in Figure 16.5: each circle represents a single Xitami server process; the ellipses are LRWPs, labeled with the name of the application they are serving; and a dotted line surrounds the host. This makes it easy to serve the same applications at different ports—and your LRWPs can serve the same or different data sources on different ports or IP addresses. That is just a matter of the initialization code. This gives you a single-host architecture in which you can apply parallelism at the web level (by replicating servers), and at the application level (by replicating LRWPs) to scale up to the level of performance you need. The different server instances can serve webs with different structures, allowing a single computer to host several web sites with a degree of flexibility.

Even this is not the end of the scalability, however: Xitami allows you further room to grow when you have exhausted the resources of a single host. All intercomponent communications in Figure 16.5 (ISAPI extensions, limited to local access, are ignored in the diagram) are network sockets. This means that they could just as easily be running over a LAN, communicating between independent hosts. So a further level of scalability and flexibility can be achieved by extending the architecture to span several machines. In this case, LRWPs can be either local or remote.

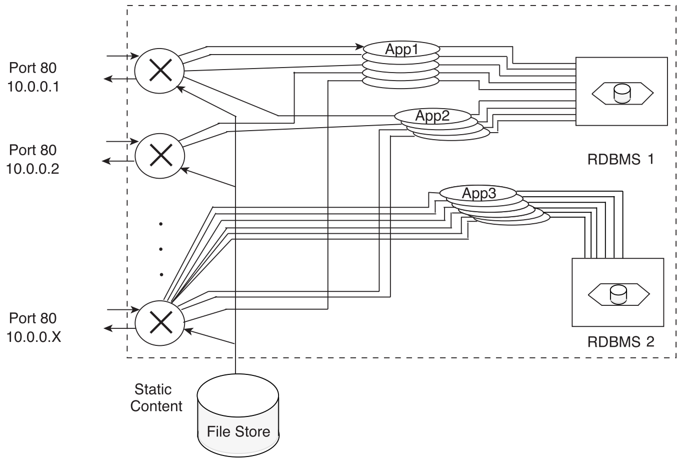


Figure 16.5 Several Xitami servers running on different IP addresses on a single host.

As Figure 16.6 shows, you could then locate your LRWPs where you want them—for example, because they accessed a database for which you had no network drivers—but serve them from any of the machines running Xitami. This multihost architecture is almost frightening in its simplicity and economy. Of course the best part from our point of view is that all the serious application work can now be performed in Python-based CGI scripts and LRWPs. Although Figure 16.6 shows a separate host for each web, application, and database, there is no reason why you cannot partition the tasks for best use of any particular hardware combination. Putting each process on a separate host would probably be a wasteful strategy.

A Simple, Flexible LRWP-Based Application

To give you the flavor of writing database code behind LRWPs, I have put together an example that runs nicely with Xitami. It uses a database to store most of the content, although text templates determine the look-and-feel. This is a slightly error-prone method of generating HTML, but it shows how you can easily achieve stylistic uniformity. You will learn about more structured methods of content generation later in the chapter.

Database Structure

The database structure used is simple: there is one table called `StdPage`, and each row in the table represents a page in the database. The pages are presented as a linked set on the home page and as links in the left-hand navigation bar on the standard page layout. Table 16.2 illustrates the structure.

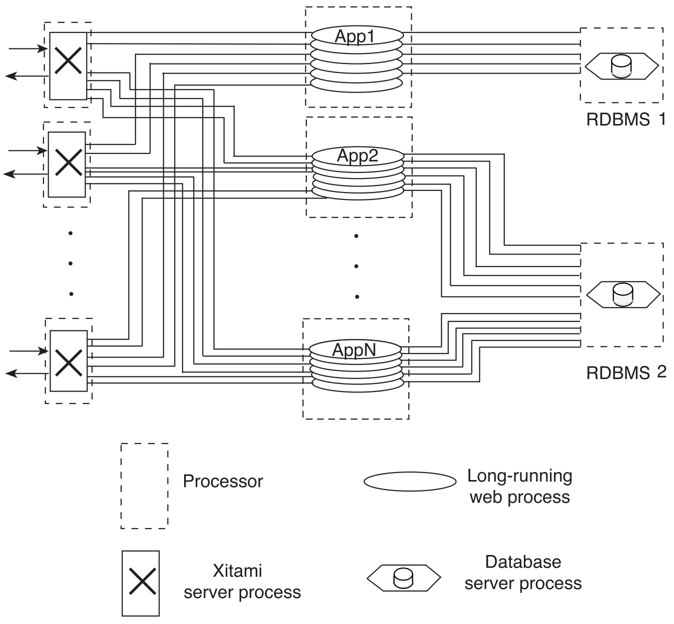


Figure 16.6 Xitami architecture with functions on individual processors.

Table 16.2 **Layout of the *StdPage* Table That Drives the *Xitami1* Database**

Field Name	Data Type	Description
`Name`	Text	Code used as primary key.
`CrDT`	Date/Time	Creation timestamp.
`MoDT`	Date/Time	Modification timestamp.
`Num`	Number	Defines the ordering of the pages.
`PageSet`	Text	We are only interested in type `Std`.
`LinkText`	Text	Text used in linking and titling the page.
`Template`	Text	`None`, or alternative page style.
`Content`	Long Text	What the user actually sees.

To make the project a little more interesting, you can edit the pages of this web (in other words, the content of the database that generates them) by clicking on a link at the bottom of each of its pages. Only the home page is fixed in content. To run this server, you must have a functioning copy of Xitami running, and you will need to register the `HoldenWebSQL` data source (if you are using ODBC) or modify the database access code (if you want to use some other module for database access). The book's web site contains a copy of the database used in testing, which you can either use with the Jet driver on Windows or migrate to the engine of your choice on your preferred platform.

The architecture of this application repays some study, and you can run several copies of the LRWP against a single Xitami server to determine whether this improves the application's performance. A modified version of the client program from Listing 16.4 might be helpful in load testing but is left as an exercise.

*Application Server Module (*appsrv.py*)*

This section presents the code of the long-running web process that implements the basic service I call a *mini-web server*. For each request it receives, this LRWP calls a page generation routine selected according to the `page` argument in the URL query string. The page generation logic, being contained in other modules, is described in later sections.

```
 1  #! D:/Python20/python
 2
 3  import mx.ODBC.Windows, cgi, sys
 4  import lrwplib as web
 5  import stdpage
 6  import frmpage
 7
```

This module (which lives with the other code in the `xitami1` subdirectory of the web site Chapter 16 code) uses a database and the CGI interface. It is a long-running web process, and it uses two local modules to generate standard and forms-based pages for the web client.

```
 8  #
 9  # One-time LRWP startup logic
10  #
11  try:
12      db = mx.ODBC.Windows.Connect("HoldenWebSQL")
13      CR = db.cursor()
14      print "Connected to database"
15  except:
16      sys.exit("Could not connect to database.")
17  try:
18      lrwp = web.LRWP("xitami1", '127.0.0.1', 81, '')
19      lrwp.connect()
20      print "Connected to Xitami"
21      sys.stdout.flush()
22  except:
23      sys.exit("Could not start long-running web process.")
24
```

Lines 8 through 24 are standard LRWP startup logic, connecting to the database and the Xitami server. Remember that you need to change the port number if you configured Xitami with a port base number other than zero. The LRWP seems to work well with several DB API-compliant driver module and engine combinations.

```
while 1:
    #
    # Per-request code
    #
    request = lrwp.acceptRequest()
    query = request.getFieldStorage()

```

The application server then begins an infinite loop, which starts by getting a request from Xitami and building a `FieldStorage` structure from the query string.

```
    #
    # Page generation logic
    #
    request.out.write("""<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2➥
    Final//EN">
    <html>
    <head>
    <title>Xitami LRWP: Simple Sample App Server</title>
    </head>
    <body>
    """)

    try:
        PageName = query["page"].value
    except KeyError:
        PageName = "default
```

With an unsophisticated approach to content generation, the application simply assumes that all output will be HTML pages with the same title. This is not a brilliant practice, especially since it will confuse people trying to use their browser history when they see a list of pages all called "Xitami LRWP: Simple Sample App Server"! The LRWP retrieves the name of the page from an argument to the URL, so it can generate the required content. If it finds no page name, it reverts to its default page, a generic home page with links to each standard page in the table.

```
#
# XXX This should really be done by some more elegant
# technique, but we are in hack mode just now...
#
    if PageName == "default":
        r = stdpage.default(query, request, CR)
    elif PageName == "StdPage":
        r = stdpage.StdPage(query, request, CR)
    elif PageName == "FormStdPage":
        r = frmpage.FormStdPage(query, request, CR)
    elif PageName == "NewStdPage":
        r = frmpage.NewStdPage(query, request, CR, db)
    elif PageName == "UpdateStdPage":
        r = frmpage.UpdateStdPage(query, request, CR, db)
    else:
        r = "<h2>Page '%s' not found...</h2>" % (PageName, )
    request.out.write(r)
    request.out.write("""
</body>
</html>
""")
    request.finish()
    if PageName == "Finish":
        break

```

Calling the appropriate page function generates the content. The preceding code is a straightforward and inelegant way of calling the right function for a particular page type. Note that the server passes the query and the request through to the page content generator functions: it has no idea which page is to be produced, and it does not care. There is a "back door" to allow easy termination of the LRWP by setting the `Page` argument to `Finish`. It should be eliminated from a production server because the last thing you want to do is stop the server on the instructions of J. Random User.

```
#
# LRWP Process termination
#
db.close()
lrwp.close()
```

If a URL containing `Page=Finish` as an argument arrives, the server tidies itself up and terminates. Of course the logic you have just seen implements only the main application's flow of control. It uses the page generation modules to build the HTML content that the user sees rendered in the browser window. The annotated source from the `stdpage` module follows.

*Standard Pages Module (*stdpage.py*)*

Standard pages are those containing only simple HTML content. These pages are conveniently generated by functions defined in the same module because that allows them to easily share a common look-and-feel, implemented by module-level functions.

```
import urllib
from dtuple import TupleDescriptor, DatabaseTuple

StdPageBody = open("text/StdPageBody.txt").read()
DefaultBody = open("text/DefaultBody.txt").read()

```

The code uses the `dtuple` module to make it easier to refer to database fields by name in the code. It reads in the standard page body templates from a text file. You can use a graphical editor such as FrontPage to generate the look-and-feel if you want, just dropping in `%s` strings where the LRWP should insert particular pieces of content retrieved from the database.

```
#
# Left-Hand Navigation Bar
#
def LHNav(ThisPage, CR):
    dt = TupleDescriptor([["Name"], ["LinkText"], ["Template"]])
    CR.execute("""SELECT Name, LinkText, Template FROM StdPage
                    WHERE PageSet='Std' ORDER BY Num""")
    SP = CR.fetchall()
    LHNavBar = []
    for sp in SP:
        sp = DatabaseTuple(dt, sp)
        if sp.Template == "None":
            tname = "StdPage"
        else:
            tname = sp.Template
        if sp.Name != ThisPage:
            mgtext = """<img src="/HWimages/cblob.gif" border=0 alt=""➥
            width=10 height=8>
                        <a href="/xitami1/?page=%s&Name=%s">%s</a>""" %➥
                        (tname, urllib.quote(sp.Name), sp.LinkText)
        else:
            mgtext = """<img src="/HWimages/rblob.gif" width=10 height=8➥
            border=0 alt="">""" + str(sp.LinkText)
        LHNavBar.append("<nobr>%s</nobr><br><br>" % (mgtext,))
    LHNavBar = "".join(LHNavBar)
    return LHNavBar

```

The left-hand navigation bar function retrieves three fields for each of the standard pages in the database. It iterates over the retrieved rows, generating a suitable hypertext link for each (except the current page: in this case, a visual flag is used to indicate that it *is* the current page). If a particular page requires a non-standard page style, this can be flagged in the `Template` column for the page. This feature is not currently used and is not yet implemented elsewhere in the code. It is, however, a convenient way of introducing variety in the appearance of pages generated from a common database.

```
#
# Default site page
#
def default(form, req, CR):
    CR.execute("""SELECT Name, LinkText, Template FROM StdPage
            WHERE PageSet='Std' ORDER BY Num""")
    SP = CR.fetchall()
    ct = 0
    FtrNavBar = []
    for sp in SP:
        template = sp[2]
        if template == "None": template = "StdPage"
        FtrNavBar.append("""<nobr>
        <img src="/HWimages/cblob.gif"><a➥
        href="/xitami1/?page=%s&Name=%s">\%s</a></nobr>&nbsp;""" %➥
                (template, urllib.quote(sp[0]), sp[1]))
        ct = ct + 1
        if ct == 8:
            ct = 0
            FtrNavBar.append("<BR>")
    return DefaultBody % ("".join(FtrNavBar), )

```

The home page for the site uses the `DefaultBody` template, read when the module was initialized. Because it needs to refer to each page, it again uses the database to retrieve the required information. The link generation here is not so smart and needs updating to use the `Template` column in the same way as the left-hand navigation bar. It also would be aesthetically more pleasing to distribute the links evenly across the lines instead of simply breaking after every eight links.

```
#
# Page Body Builder (XXX UNSOPHISTICATED)
#
def PageBody(LinkText, LHNav, Content, EditLink=None):
    if EditLink:
        Link = """<font size="-3"><center><A HREF="?page=%s">EDIT
        THIS PAGE</a></center></font>""" % (EditLink, )
    else:
        Link = ""
    return StdPageBody % (LinkText, LHNav, Content, Link)

```

All standard page routines call the `PageBody()` function to generate the content they return. Up to four variable elements are passed as arguments. Certain pages are generated "on-the-fly," and these do not provide an `EditLink` argument (since they cannot be edited).

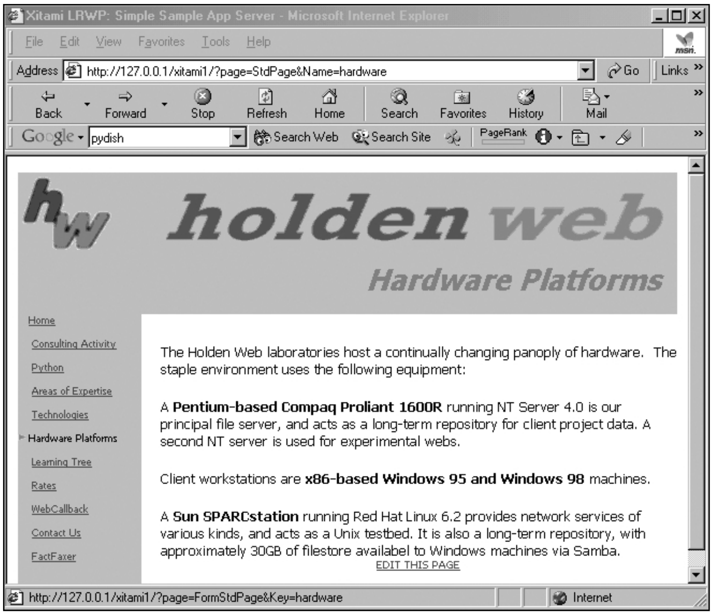
```
#
# Standard Page
#
def StdPage(form, req, CR):
    try:
        StdPageName = form["Name"].value
    except KeyError:
        return "<h2>Name not given for standard page display</h2>"
    # XXX lines below should be replaced with DatabaseTuple
    FldList = (['Name'], ['LinkText'], ['PageSet'], ['Num'], ['Content'])
    dt = TupleDescriptor(FldList)
    CR.execute("""SELECT Name, LinkText, PageSet, Num, Content
            FROM StdPage WHERE  Name=? AND PageSet='Std'""",
                            (StdPageName, ))
    TP = CR.fetchone()
    if TP == None:        # Use a page which should always exist
        CR.execute("""SELECT Name, LinkText, PageSet, Num, Content
                FROM StdPage WHERE Name='ERROR'""")
        CR.fetchone()
    rec = DatabaseTuple(dt, TP)
    LHNavBar = LHNav(StdPageName, CR)
    return PageBody(rec.LinkText, LHNavBar, rec.Content,➥
    "FormStdPage&Key="+StdPageName)
```

Finally comes the function to generate standard page contents. It extracts the name of the page from the URL if one was given (if not, it simply returns a message to the user, who will doubtless be perplexed). The page name is used to pull the required content from the database, and if the specified page is not found, then a standard `ERROR` page is pulled from the database instead. If this page is missing, the logic fails horribly. The database fields are then used to generate the page content, which is returned to the application server. Figure 16.7 shows a browser window with a page from the database displayed.

Although the screen dump for Figure 16.7 omitted the cursor, it was actually positioned over the `EDIT THIS PAGE` link at the foot of the page. You can see in the status bar that this is a hypertext link to the following URL:

```
http://127.0.0.1/xitami1/?page=FormStdPage&Key=hardware
```

If you click on this link, then you start to invoke pages from the `frmpage` module, whose listing you will find in the next section.


Xitami LRWP: Simple Sample App Server - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://127.0.0.1/xitami1/?page=StdPage&Name=hardware
Go Links
Back Forward Stop Refresh Home Search Favorites History Mail
Google pydish Search Web Search Site PageRank
holden web
Hardware Platforms
Home
Consulting Activity
Python
Areas of Expertise
Technologies
Hardware Platforms
Learning Tree
Rates
WebCallback
Contact Us
FactFaxer
EDIT THIS PAGE
http://127.0.0.1/xitami1/?page=FormStdPage&Key=hardware
Internet

The Holden Web laboratories host a continually changing panoply of hardware. The staple environment uses the following equipment:

A **Pentium-based Compaq Proliant 1600R** running NT Server 4.0 is our principal file server, and acts as a long-term repository for client project data. A second NT server is used for experimental webs.

Client workstations are **x86-based Windows 95 and Windows 98** machines.

A **Sun SPARCstation** running Red Hat Linux 6.2 provides network services of various kinds, and acts as a Unix testbed. It is also a long-term repository, with approximately 30GB of filestore availabel to Windows machines via Samba.

Figure 16.7 A standard `appsvr` page displayed by Internet Explorer.

***Form Pages Module* (frmpage.py)**

```
from dtuple import TupleDescriptor, DatabaseTuple
import pyforms
import pySQL
import stdpage

```

This module in turn uses two others, `pyforms` and `pysql`, for which there is no detailed discussion in this chapter. You covered the basics of form generation in Chapter 11, "Adapting the Python Interface to Database Products," and the generation of SQL statements is rather unsophisticated. If you have any questions, use the source, Luke! Bear in mind as you do so that these were early implementation prototypes, so I do not recommend them as examples of best practice. They do perform as required, however, and you might find them useful (if a little inflexible) in other contexts. Functions from the `stdpage` module generate simple "on-the-fly" pages for error messages.

The first function in this module uses a form generator driven by a form description to build the page content. The form is populated from the database or with blank data depending on the URL that triggered it.

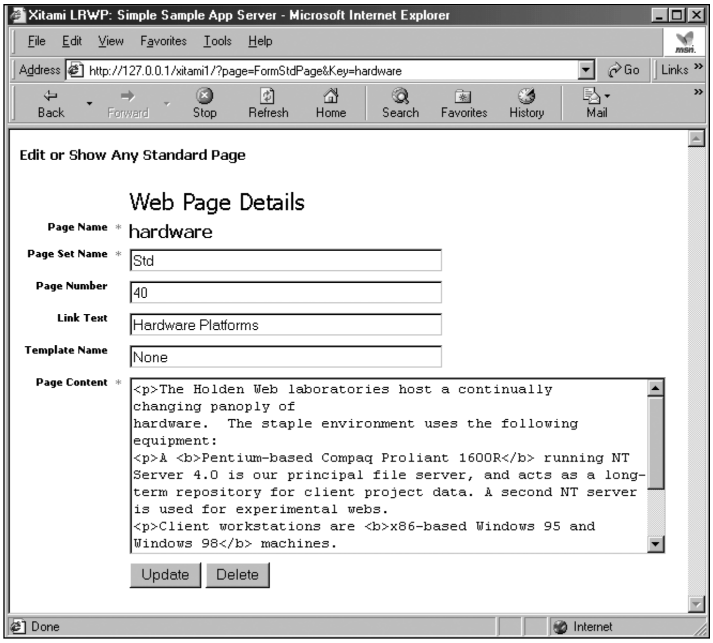
```
#
# Form for Standard Page Update
#
def FormStdPage(form, req, CR):
    result = [ """
    <h4>Edit or Show Any Standard Page</h4>
    <table cellpadding="2">

    """]

    try:
        KeyVal = form["Key"].value
    except KeyError:
        KeyVal = None
    if KeyVal == None:    # Bare call to page gets list of pages
        Stmt = "SELECT Name, LinkText, Num from StdPage WHERE PageSet =➥
        'Std' ORDER BY Num"
        CR.execute(Stmt)
        records = CR.fetchall()
        dt = TupleDescriptor((['Name'],['LinkText'],['Num']))
        for r in records:
            RS = DatabaseTuple(dt, r)
            result.append('<tr><td>%s</td> <td>%s</td><td>%d</td><td><a➥
            href="?page=FormStdPage&Key=%s"> Edit</a></td><td><a➥
            href="?page=StdPage&Name=%s"> Show</a></td></tr>\r\n'➥
                            % (RS.LinkText, RS.Name, RS.Num, RS.Name,➥
                            RS.Name))
        result.append("""
        </table>
31          <BR><a href="?page=FormStdPage&Key=*CREATE*"> ***CREATE NEW➥
            RECORD***</a><BR>\r\n""")
32      else:
33          if KeyVal == "*CREATE*":
34              RS = None
35          else:
36              Stmt = "SELECT Name, PageSet, Num, LinkText, Template, Content➥
                FROM StdPage WHERE name=?"
37              CR.execute(Stmt, (KeyVal, ))
38              record = CR.fetchone()
39              dt = TupleDescriptor((['Name'],['PageSet'],['Num'],➥
                ['LinkText'],['Template'],['Content']))
40              RS = DatabaseTuple(dt, record)
41          Flist = (
42              # Note options MUST now be a string, not None
43              ["Comment1", "Web Page Details", "REM", "+2", ""],
44              ["Name", "Page Name", "KSA", 20, "R"],
45              ["PageSet", "Page Set Name", "T", 20, "R"],
46              ["Num", "Page Number", "N", 5, ""],
47              ["LinkText", "Link Text", "T", 30, ""],
48              ["Template", "Template Name", "T", 30, ""],
49              ["Content", "Page Content", "M", (10, 60), "R"]
50          )
51          result.append(pyforms.FormBuild(Flist, "StdPage", "name",➥
            str(KeyVal), RS))
52      return "".join(result)
53
```

This is a multifunction page. If it is called without specifying the page to be edited (that is, the URL contains no `Key` argument), then lines 21 through 31 generate a list of the pages in the database, each linked with the `Page` and `Key` arguments to edit that page. A link to display each page's content also is included, to allow users to check what they are going to edit and to return easily to the main web content.

If the URL included a `Key` argument, then lines 33 through 51 generate a form, populated with the contents of the specified database record. The argument value `"*CREATE*"` is an instruction to display an empty form instead. Here most of the work is done by the `pyforms.FormBuild()` routine, driven by the form description bound to the `Flist` variable. Figure 16.8 shows a browser displaying a form, after I clicked on the `EDIT THIS PAGE` link in Figure 16.7. The form page itself could have conformed to the look-and-feel of the other pages, but this seemed unnecessary.


Xitami LRWP: Simple Sample App Server - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://127.0.0.1/xitami1/?page=FormStdPage&Key=hardware
Back Forward Stop Refresh Home Search Favorites History Mail
Update
Delete
Done
Internet

Edit or Show Any Standard Page

Web Page Details

Page Name *	hardware
Page Set Name *	Std
Page Number	40
Link Text	Hardware Platforms
Template Name	None

Page Content *

```
<p>The Holden Web laboratories host a continually
changing panoply of
hardware.  The staple environment uses the following
equipment:
<p>A <b>Pentium-based Compaq Proliant 1600R</b> running NT
Server 4.0 is our principal file server, and acts as a long-
term repository for client project data. A second NT server
is used for experimental webs.
<p>Client workstations are <b>x86-based Windows 95 and
Windows 98</b> machines.
```

Figure 16.8 A dynamically generated form with data preloaded.

The form that is generated uses different actions according to its function. The form's action attribute is not specified in the `<FORM>` tag because all pages in this web have the same URL and simply differ in their `Page` and `Name` argument values. The form therefore contains a hidden field that sets the `Page` argument to `NewStdPage`. The `lrwplib getFieldStorage()` method generates the `FieldStorage` from the query string for a `GET` method call and from the standard input for a `POST`. When acting on an existing record (whether update or deletion is being requested), the form's `Page` input will contain `UpdateStdPage`, which can handle both updates and deletions (according to which Submit button was clicked). This effectively selects one action routine for page creation and a different one for update and delete.

```
54  #
55  # New Standard Page
56  #
57  def NewStdPage(form, req, CR, db):
58      try:
        Flist = eval(form["#Flist#"].value)
        KeyVal = form["#KeyVals#"].value
    except KeyError:
        return "<H4>Could not access #Flist# or #KeyVals#</H4>"
    if KeyVal != "*CREATE*":
        return """<H4>ERROR: New Called with Key Value!</h4>"""
    else:
        Stmt = pySQL.SQLInsert(form, "StdPage", Flist, "" ,"", "")
    result = ["<BR>SQL IS: %s" % (Stmt, )]
    try:
        CR.execute(Stmt)
        db.commit()
    except:
        db.rollback()
        return stdpage.PageBody("Database Operation Error",
                        stdpage.LHNav(KeyVal, CR),
                        "Sorry, unable to create this page")
    result.append("""
    <h2>StdPage %s Created</h2>
    <BR>
    <A HREF="?page=%s">Back to Page List</A>
    """ % (form["Name"].value, "FormStdPage"))
    return "".join(result)

```

The `pyforms` module actually cheats a little bit. It passes the form description (which it received as an argument) across to the processing page as a hidden field of the form, as it does with the key value. This technique is not recommended for public use. A malevolent user would be able to decode all this and modify the form description. You might improve the security by encrypting these values using a key known only to the program, or by retaining this value on the server using some kind of session-state mechanism. It would be even better practice to store the forms descriptions as a part of the database and simply pass database key values between successive pages.

Further, `pyforms` relies on the form field value to be a Python expression, which it evaluates to recover the form description. This practice is even more dangerous because the aforementioned malevolent user would be able to spoof an HTTP request that caused the server to execute arbitrary Python code!

The mechanism is convenient, of course. However, in an environment where your network's security is potentially at risk, convenience should not be a factor. If you are worried about the security implications of this code, note that you will learn solutions to these problems in Chapter 17, "AWeFUL: An Object-Oriented Web Site Framework," and Chapter 18, "A Web Application—PythonTeach.com."

The `NewStdPage` function generates a `SQL INSERT` statement using the `pySQL` library and executes it. It generates a failure page and rolls back the current transaction if the execution raises any exceptions. If it succeeds, then you can see the SQL statement in the HTML output for debug purposes. A final link allows the user to navigate back to the list of pages for more editing work or to return to page viewing.

```
#
# Update Standard Page
#
def UpdateStdPage(form, req, CR, db):
    Operation = str(form["Submit"].value)
    Flist = eval(str(form["#Flist#"].value))
    KeyVal = str(form["#KeyVals#"].value)
    if Operation == "Update":
        Stmt = pySQL.SQLUpdate(form, "StdPage", (("Name","S"),), (KeyVal,),➥
        Flist, "")
        which = "StdPage&Name=" + KeyVal
        pp = "Updated Page"
    elif Operation == "Delete":
        Stmt = pySQL.SQLDelete("StdPage", (("Name","S"),), (KeyVal,))
        which = "default"
        pp = "Home Page"
    try:
        CR.execute(Stmt)
        db.commit()
    except:
        db.rollback()
        return stdpage.PageBody("Database Operation Error",
                        stdpage.LHNav(KeyVal, CR),
                        "Sorry, unable to update this page's details")
    return stdpage.PageBody("%s Completed" % (Operation, ),
                        stdpage.LHNav(KeyVal, CR),
                        """
    <h2>Page %s&nbsp;%sd</h2>
    <!-- <BR>SQL IS: %s<BR> -->
    <BR>
    <A HREF="?page=%s">Show %s Page</A>
    """ % (KeyVal, Operation, Stmt, which, pp),
                        None)
```

The `UpdateStdPage` function handles both updates and deletions, and deduces the required function from the value of the Submit button the user pressed. The routine generates the correct SQL statement (look in `PySQL.py` if you want to know how) and runs it to update the database. Finally, it generates a page that informs the user that it has made the requested update and offers a link to the updated page.

*Forms and SQL Generation Modules (***pyforms.py** *and* **pySQL.py***)*

The only routine intended for public consumption in the `pyforms` module is `FormBuild`, which has a long list of arguments, some of which are complex in structure. When you call

```
FormBuild (List, Action, KeyNames, KeyVals, R=None
```

`FormBuild()` should return a complete HTML form. The first argument is a form description of the sort you saw in the `frmpage` module: a tuple holding a name, a description, a field type, a size, and some options represents each field in the form.

The `Action` argument is appended to either `New` or `Update` and used as the value of the form's `ACTION` attribute. This will select the appropriate processing function as described previously. `KeyNames` and `KeyVals` are two lists of equal length, passed in the form as hidden fields to allow the required database update to take place after the form is filled out (another security risk). Finally, `R` is a data row, which might have been read in from a database or pulled from a form currently being processed. For new data, you should use the default value rather than passing a data row.

The `pySQL` module defines three functions: `SQLInsert`, `SQLUpdate`, and `SQLDelete`. The argument lists for these functions are long, possibly longer than strictly necessary.

```
SQLInsert(Form, TableName, Fields, iFlds, iVals, Prefix)
```

Here the `Form` argument is the form data, which should be a `DatabaseTuple`, generated from either a `FieldStorage` object containing forms input from a web client, or from a database row.

The individual fields need only be accessible using subscripting by name, so you could use a dictionary if you chose. `TableName` is, as you would expect, the name of the database table on which to operate. The `Fields` argument is the form description as used by the forms pages: the first item of each tuple is used as a field name; other elements determine whether the field is included in the SQL statement and, if so, how it is processed.

The `iFlds` and `iVals` arguments hold the names and values of any other fields that need to be included in the `INSERT` statement, and the `Prefix` argument is a simple string you can use to select processing of only a subset of the data in the form whose names all begin with the prefix.

```
def SQLUpdate(Form, TableName, KeyNames, KeyVals, Fields, Prefix
```

This function uses the same `Form` and `TableName` arguments as `SQLInsert()`, but these are followed by a list of key names and values (which together should uniquely specify a single row of the table if just one row is to be updated, as is normally the case). The `Fields` and `Prefix` arguments again describe the form contents and determine which fields should be processed.

```
def SQLDelete(table, KeyNames, KeyVals)
```

This function generates a SQL `DELETE` statement, using arguments described previously. Again, the key names and values should ideally specify a unique row in the table, but if you know what you are doing (and if you are careful), you can use `SQLDelete` and `SQLUpdate` in other contexts to generate statements affecting multiple rows.

Architectural Weaknesses of the Xitami Sample

Probably the most obvious, and perhaps the most disappointing, thing about this server is its failure to use the object-oriented parts of Python. It defines no classes and creates relatively few objects. The procedural style of the code is easy to understand, but the execution sequence for any given page request is linear. There may be other, more efficient ways to organize this or similar code.

Because the LRWPs are not maintaining state, they are simple, and any LRWP in a set can handle any incoming request for any page in an application. When the LRWPs need a context for a particular set of page accesses, then the state data has to be stored somewhere all the processes can access it because Xitami takes no notice of state when handing out work to a set of LRWPs all registered for the same application.

The best way to manage web session-state information is for the server to return a cookie, whose value represents a session-specific key value, to the browser. When the browser sends further requests, it returns the cookie, and the server can use it to locate the state data for the browser's session. It is *not* appropriate to use cookies for large quantities of data for three reasons: first, because this will use unnecessary network bandwidth; second, because browsers place limits on the number and size of cookies they will store; and third, because the information is much less secure when transmitted over the network than when it is stored on the server.

Much of the data retrieved from the database is purely structural: it appears in navigation bars, lists of links, and so on. In most sites, this type of data is unlikely to change with any frequency, so it would make sense for the application server to cache a copy, refreshing it infrequently. No such sophistication appears here, however. To keep the code simple, everything is read from the database as it is needed.

Zope Integration with Python

Zope (`www.zope.org`) is probably the closest thing there currently is to the Python "killer app." The question of Python integration is almost trivial because Zope itself is written entirely in Python. Whenever someone asks for a list of large Python applications, Zope is always among the first to be mentioned. Zope is an object-oriented web framework whose `ZServer` core server module is built on the Medusa architecture you looked at in Chapter 7, "Asynchronous Services," but there is a huge amount more to the system. The Python integration is therefore a given, and Zope (formerly Digital Creations) employs the team who maintains the Python core.

You maintain and administer Zope content using web and/or FTP access, so you can perform these tasks as easily from a remote location as locally. It is an open-source product, so after you master its complexities, it is extremely cost-effective, and of course if you are even more ambitious, then you can extend it with your own custom code. Indeed, several books either newly published or shortly to appear, will help you make sense of this complex software. Here I will only try to sketch the briefest outline, to try to show you what Zope can do in broad terms. If you need more in-depth understanding, try *The Zope Book* by Amos Latteier and Michael Pellatier (© 2002 New Riders).

Zope is based around a transactional object database, ZODB, maintained by Andrew Kuchling. This database allows you to specify a set of changes as a single transaction and to undo them all as a whole rather than individually. Each object in the database maps to a URL, and vice versa, reflecting the underlying philosophy that the web is object-oriented.

You specify the content of a Zope site largely in DTML, an HTML-like tagged markup language. DTML allows you to safely script your content based on the Zope objects available in the context of the current URL. Because the content is in a hierarchy, it is possible to write a DTML description of your site's look-and-feel at the top of the tree and have it cascade down to affect all pages that do not explicitly override it. Security of a fine-grained nature applies to all Zope objects, and again a security setting established at the top of the tree will cascade downward unless overridden. This general capability to build the environment from the top down is called *acquisition* in Zope terminology.

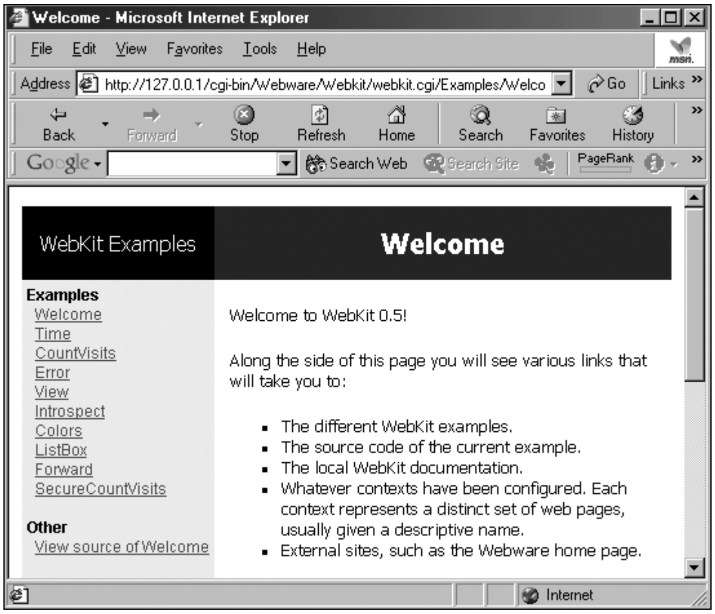
Acquisition is similar in principle to the concept of inheritance as implemented in object-oriented systems but rather different in practice. The relationship between a class and its subclasses is statically determined, whereas acquisition relationships are determined by containment. Zope objects can implement services, and when objects are placed inside a folder, the folder acquires the capability to provide the services implemented by the objects, and the objects acquire all the services of the folder. Because folders are also Zope objects, they acquire services from their containing folders, in the same way as subclasses inherit methods and attributes from their base classes.

Other objects in the Zope object tree can be SQL methods, which specify sets of relational data. These too can be used in DTML, giving good integration with external data sources. You can also drop `Zcatalog` objects into the object tree, which enable indexing of arbitrary content underneath the object whether it is relational data, DTML, email messages, or LDAP data.

To facilitate development, there can be several *Versions* of a given Zope system, which are independently maintained in the object database. This means that several developers can each be working on their own Version without affecting other developers or the user community. The Version can thus be debugged and tested for stability in isolation from other changes that may be taking place. When everything is approved, the Version can be checked in, making it visible to the user community at large.

Webware

Webware (`http://webware.sourceforge.net/`) is a Python web application suite designed by Chuck Esterbrook and others to help produce web applications that are object-oriented, cached, and multithreaded. The components are designed to work together, although some of them can be used independently, and the package typically uses standard technologies such as servlets and server pages, which are already familiar to many web developers. All components are open-source, and the project has spent a lot of time ensuring that the documentation is helpful and to the point. The installation was simple (after a fight with a recalcitrant path setting), and using Xitami as the base web server, I soon saw the WebKit page shown in Figure 16.9.


Welcome - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://127.0.0.1/cgi-bin/Webware/Webkit/webkit.cgi/Examples/Welco
Go Links
Back Forward Stop Refresh Home Search Favorites History
Google Search Web Search Site PageRank
Internet

WebKit Examples

Welcome

Examples
Welcome
Time
CountVisits
Error
View
Introspect
Colors
ListBox
Forward
SecureCountVisits

Other
View source of Welcome

Welcome to WebKit 0.5!

Along the side of this page you will see various links that will take you to:

- The different WebKit examples.
- The source code of the current example.
- The local WebKit documentation.
- Whatever contexts have been configured. Each context represents a distinct set of web pages, usually given a descriptive name.
- External sites, such as the Webware home page.

Figure 16.9 A WebKit page soon after Webware installation.

Among the nicer features of the online documentation is the object hierarchy listing, which contains links to a summary of, and to the complete source for, each class.

WebKit is the central component of Webware, and it allows the Python programmer to work with applications, servlets, requests, responses, and transactions. The application is a server-side entity, which receives incoming requests and delivers them to servlets. The servlets in turn produce responses, which the server returns to the client. Transactions hold references to all the other types of objects and are accessible to them all. The application therefore bears some resemblance to a Xitami long-running web process. After you have installed Webware, you must run the application server before the Webware components can be activated.

In the simplest installations, and for testing, your requests are passed to the application server by the `webkit.cgi` script. As well as the standard CGI adapter, which is the easiest way to get Webware started, there are also adapters for FastCGI, `mod_python`, and `mod_snake` as well as a native `mod_webkit` for Apache. Unlike CGI scripts, these adapters stay resident in memory, providing a significant performance increment. There is currently no LRWP adapter, which would be a nice enhancement for Xitami users.

To give you the flavor of Webware pages, Listing 16.6 shows you the source of a secure page counter. This defines a subclass of `SecurePage`, which requires only the `writeContent()` method to complete its functionality.

Listing 16.6 **A Secure Page Counter in Webware**

```
from SecurePage import SecurePage

class SecureCountVisits(SecurePage):
    def writeContent(self):
        count = self.session().value('num_visits', 0)+1
        self.session().setValue('num_visits', count)
        if count>1:
            plural = 's'
        else:
            plural = ''
        self.writeln("<p> You've been here %d time%s." % (count, plural))
        self.writeln('<p> This page records your visits using a session➥
        object. Every time you RELOAD or revisit this page, the counter will➥
        increase. If you close your browser, then your session  will end and➥
        you will see the counter go back to 1 on your next visit.')
        self.writeln('<p> Try hitting RELOAD now.')
        user = self.getLoggedInUser()
        if user:
            self.writeln('<p> Authenticated user is %s.' % user)
        self.writeln('<p> <a href="SecureCountVisits">Revisit this page</a>➥
                 <a href="SecureCountVisits?logout=1">Logout</a>')
```

Among other components included with Webware are

- **PSP (Python Server Pages)**—Allows you to use an ASP-like syntax to mingle Python code and HTML. (PSP is described later in this chapter.)
- **ComKit**—Lets you use COM objects in a multithreaded web context, giving access to a wide range of ActiveX components.
- **TaskKit**—Gives you tools to schedule tasks at repeated intervals, or at specific times.
- **MiddleKit**—For building the "middle tier" (the application-specific objects implementing business rules) of multitier applications, which communicates with both the web front end and a back-end data repository.
- **UserKit**—Handles the management of a user base, including their passwords and related data.
- **CGI Wrapper**—Allows you to easily import "legacy" CGI code into your Webware environment.
- **WebUtils**—Includes common functions for a variety of web-related tasks.

The major feature of the Webware application server is its capability to dispatch web requests to be handled by Python objects. In this respect, it closely parallels the Zope architecture; although it is more lightweight, the coupling between its components is looser, and it is therefore easier for Python programmers to dip into.

More advanced developers may enjoy Webware's "plug-in" architecture and servlet factories, which make extending Webware feasible. An increasing number of plug-ins and patches, accessible via the Webware home page and mailing list, have sprung up over time. There is also built-in support for servlets that handle XML-RPC, and the design allows for the addition of protocols besides HTTP. The product is supported by an active developer group and has been used to support a number of professional intranet and public web sites already. It is one of the more hopeful signs in the current Python web server world.

HTML (and XML) Generation

There are as many ways to generate HTML in Python as there are to persuade Python scripts to interact with a web server. Some approaches are more complete, some better known, than others. A first stopping point might be `http://www.python.org/topics/web/HTML.html`, which lists some possibilities but by no means all.

The Python world is full of projects that start, show promising results, and then run out of steam before all the loose ends are tied up and the whole packaged up for convenient use. When it comes to HTML generation, the situation is no different, with some initially promising packages still in alpha after two years lying fallow and release numbers between 0.1 and 0.5 quite common. I suspect that sometimes this is due to the 80/20 rule, which suggests that you can get 80 percent of the return on a project for 20 percent of the investment. When you are writing open-source code, the last 20 percent can seem not worth four times the effort to date, particularly if other projects are further along.

Whether there is any way to stop such duplication of effort is another question. Web sites such as SourceForge and FreshMeat do a good job of publicizing projects in need of assistance, but many other projects are undertaken independently. At least those who start a project but do not complete it have learned a lot about writing software, besides picking up some programming technique. It takes more than just a good programmer to bring a project to completion.

Despite these gloomy ponderings, there are some exceptionally bright spots in the HTML landscape and some approaches you should know about whether you intend to adopt them or choose a "roll-out-your-own" approach. Because of their diversity it is only possible to pick a representative sample. You will find many other tools on the web that do similar things, and the ones you read about here are simply the best-known.

Programmed Techniques

The most flexible way to generate HTML is by writing code (Python, naturally), so this first section outlines the most popular way of doing that. These systems are equally suitable for producing static content but are not limited to that role.

HTMLgen

This module (found at `http://starship.python.net/crew/friedrich/HTMLgen/html/main.html`) is probably the best-known way of generating HTML. It defines an object hierarchy that closely parallels the hierarchy of HTML elements. Each object is capable of rendering itself in HTML, which it does when you call its `__str__()` method. Because the `print` statement also calls this method, you can generate a whole document by assembling components, appending the components to the document, and printing the document. This outputs the required HTML. Here is a simple interactive example:

```
>>> import HTMLgen
>>> lst = HTMLgen.List()
>>> lst.append("hello")
>>> lst.append("everybody run!")
>>> lst.append("goodbye")
>>> print lst
<UL>
<LI>hello
<LI>everybody run!
<LI>goodbye
</UL>
```

The general approach is to create some sort of document and then append content to it until you have assembled the whole page. You can build the content in chunks, in the same way as the preceding list. HTMLgen also allows you to use several different base objects for your document, including a `SeriesDocument`. This last is an interesting object because you can parameterize the characteristics of a whole series of documents in an initialization file and even link the set with pointers to "up," "previous," "next," and "home" documents of every page. This makes it easy to build coherent document sets composed of static content.

Albatross

Albatross (`http://www.object-craft.com.au/projects/albatross`) is a small open-source toolkit for constructing highly stateful web applications. The toolkit includes an extensible HTML templating system similar to Zope DTML. Although templates can be used standalone, Albatross also provides an application framework. A number of different mixin classes enable applications to be deployed either as CGI programs or, to improve efficiency, via `mod_python`. Application state can be stored either at the server or client. Albatross uses `distutils` for installation and provides a reasonably complete programming guide that covers all toolkit features.

Listing 16.7 is a simple CGI program that uses Albatross templates to display the CGI process environment. It creates a `SimpleContext` object in line 5 and on line 8 makes the `os.environ` dictionary available to the template under the name `environ`.

Listing 16.7 **A Simple Albatross CGI Script**

```
 1  #!/usr/bin/python
 2  import os
 3  import albatross
 4
 5  ctx = albatross.SimpleContext('.')
 6  templ = ctx.load_template('showenv.html')
 7
 8  ctx.locals.environ = os.environ
 9
10  templ.to_html(ctx)
11  print 'Content-Type: text/html'
12  print
13  ctx.flush_html()
```

The templating system uses the `locals` member of the execution context as the local namespace for evaluating Python expressions. You can see in Listing 16.8 that the template iterates over the `os.environ` dictionary, which is accessible inside the template as a result of line 8 of Listing 16.7.

Listing 16.8 **Accessing Execution Context in an Albatross Template**

```
<html>
 <head>
  <title>The CGI environment</title>
 </head>
 <body>
  <table>
   <al-exec expr="keys = environ.keys(); keys.sort()">
   <al-for iter="name" expr="keys">
    <tr>
     <td><al-value expr="name.value()"></td>
     <td><al-value expr="environ[name.value()]"></td>
    <tr>
   </al-for>
  </table>
 </body>
</html>
```

When you use Albatross to build an application, the execution context becomes the session object, and each page is implemented by a Python page module (or object) plus one or more template files. Each page module contains a `page_process()` function to handle browser requests and a `page_display()` function to generate the response. Listing 16.9 is the `login.py` page module from the Albatross `popview` example application.

Listing 16.9 ***login.py* from Albatross's *popview* Example**

```
import poplib

def page_process(self, ctx):
    if ctx.req_equals('login'):
        if ctx.locals.username and ctx.locals.passwd:
            try:
                ctx.open_mbox()
                ctx.add_session_vars('username', 'passwd')
            except poplib.error_proto:
                return
            ctx.set_page('list')

def page_display(self, ctx):
    ctx.run_template('login.html')
```

Ad Hoc Methods

If you do not have too much HTML to generate, and it is not important to be able to update the style of a set of pages with a single change, the techniques used in previous web examples in the book might be acceptable. Just parameterize HTML snippets with Python formatting symbols and use the string formatting (`%`) operator to insert the variable content you need. This technique certainly works for small webs, but the larger the web and the greater the amount of content, the more difficult this approach is to maintain. You need to be systematic if your approach is to scale up to large webs, so you should usually reserve *ad hoc* methods for experimentation, or for smaller systems where the work does not become too tedious.

Templating Tools

Sometimes there is simply too much detail in the HTML of a site to program it directly. This certainly applies when you want to take advantage of the visual effects of the newest WYSIWYG HTML generators but insert your own content inside the frameworks they produce. For these and other reasons the technique of filling in a template with variable content is a popular one, and many authors have produced useful results this way.

Cheetah

Cheetah (`http://www.cheetahtemplate.org/`) is an interesting template utility that can be used with equal facility to generate HTML, XML, PostScript, and any number of other formats. Four main principles guide its design:

- Easy *separation* of content, code, and graphic design
- Easy *integration* of content, code, and graphic design

- Limit complexity, and use programming for difficult cases
- Easily understood by non-programmers

The simple example from the Cheetah documentation shown in Listing 16.10 gives you the idea.

Listing 16.10 **A Simple Cheetah HTML Definition**

```
<HTML>
<HEAD><TITLE>$title</TITLE></HEAD>
<BODY>

<TABLE>
#for $client in $clients
<TR>
<TD>$client.surname, $client.firstname</TD>
<TD><A HREF="mailto:$client.email">$client.email</A></TD>
</TR>
#end for
</TABLE>

</BODY>
</HTML>
```

You can see that the dollar sign triggers variable substitution. Cheetah calls strings with these leading dollar signs *placeholders*. The placeholders are arbitrary Python expressions (with some syntax modifications to adapt to non-programmer use), which Cheetah evaluates when it renders the template. In very broad terms, a Cheetah template takes a definition such as the one in Listing 16.10 and a namespace (which is a Python dictionary). When you tell the template to render, it uses the namespace as the evaluation context for any placeholders it comes across in the definition. As with the UNIX shell, the placeholder following the dollar sign can be surrounded by braces to separate it from surrounding text if no spaces can be inserted.

An interesting design decision, based on the requirement that Cheetah be comprehensible to non-programmers, was to represent dictionary lookup by name qualification. Thus, if the namespace contained a variable client, which was bound to a dictionary such as `{"firstname": "Steve", "surname": "Holden", "email": "sholden@holdenweb.com"}`, then the previous example would access the elements of the dictionary. You can still use subscripting if you want, but most non-programmers find the qualified name notation much easier to work with.

For more complex data structures, the namespace used in a template can actually be a *search list*, which is a sequence of namespaces that Cheetah searches in turn until the required definition is found. This simple technique allows the creation of templates with a hierarchy of namespaces. Further, a placeholder can evaluate to another template, which is then itself evaluated, allowing Cheetah templates to be easily nested inside one another.

As you can observe in Listing 16.10, you can also include *directives* preceded by a pound sign. Some of these are Python statement constructs, interpreted much as you would expect. Others are specific constructs not related to Python. There are also macro definition facilities, which are beyond the scope of this short description. These features give you access to Python for anything that is too complex to perform using the standard placeholder features.

The Webware project has adopted Cheetah as its standard for template substitution of content. Although the software is still in beta at the time of writing, it is already stable and easy to use either with or without Webware. It looks as though Cheetah will be useful to many web projects, as well as any others that need flexible template substitution of a general nature.

Python Server Pages (PSP)

Confusingly, there are at least two systems going by the name of *Python Server Pages*, which should be no surprise. The version by Kirby Angell at `http://www.ciobriefings.com/psp/` comes with a long license I did not have time to read and appears to combine Jython (or is it *really* JPython) with a servlet engine.

The *PSP for Webware* engine at `http://writewithme.com/Webware/PSP/Documentation/PSP.html`, whose primary author is Jay Love, declares itself on its home page to be open-source and therefore commanded attention rather more immediately. From now on, this is what I refer to when I mention PSP, but I cannot determine whether the name has any trademark status.

PSP processes three special constructs:

- **`<%@ ... %>`—Directives** `import` allows you to import other Python modules. `extends` defines the base class of the current page, in the spirit of Java. `method` is used to define new methods of the class being defined by the current file. Various other directives are concerned with Webware organization, and, finally, `include` is pretty obvious.
- **`<%= ... %>`—Expressions** In a similar vein to ASP, the engine replaces such constructs with the value of the bracketed expression.
- **`<% ... %>`—Script blocks** These can span several lines, and three reserved variables are defined for use inside the blocks: `res` the response object, including a `write()` method; `req` the request object, containing web inputs; `trans` the transaction object, which provides a context for successive Webware page interactions handled by the same servlets.

Because PSP actually generates a Webware method body, normal text (not a directive, expression, or script block) gets wrapped in code that sends it out through the server. The code in script blocks is copied verbatim into the method. There is a special `<end>` tag you can use to close an indented Python suite that spans multiple script blocks.

Yaptu

Yaptu, by Alex Martelli, (available at URL `http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/52305`) stands for "Yet Another Python Templating Utility." Like Cheetah, it is not specifically geared to HTML, but rather suited to any kind of text-based document (for example, RTF, LaTeX, even plain text, as well as HTML and so on). Unlike Cheetah, Yaptu does not alter Python syntax. Yaptu lets the user define which specific regular expressions denote a placeholder (an arbitrary Python expression) and the beginning, continuation, and end of arbitrary Python statements. This ensures that Yaptu markup can indeed be used for any kind of document, since the regular expressions can be chosen to avoid any conflict with the syntax of the language of that document.

Yaptu's main claim to fame is that, despite its generality, net of comments, and docstrings, it comprises just 50 lines of code—lightweight enough to carry around wherever needed. This requires a compromise: if a Python clause to be embedded as Yaptu markup ends with a colon (such as an `if condition:`, for example), it must not be followed by a comment; if a clause does not end with a colon (such as a `break` statement, for example), it *must* be followed by a comment. If such restrictions, or some aspect of Python syntax, should prove unacceptable for a given application, Yaptu supplies two "hooks" making it easy for the application programmer to tailor things further. The application programmer can pass a "preprocessor" routine that gets a chance to manipulate all Yaptu-markup strings (expressions and statements) before Python sees them; and, for expressions only, a "handler" routine, that gets a chance to return a result and continue the templating process if an exception is raised while evaluating an expression.

Together with the capability to supply Yaptu with an arbitrary mapping to be used for "variable values" and arbitrary objects playing the roles of "input" (sources) and "output" (targets), Yaptu packs quite a punch for such a small and simple utility. It is instructive to examine the richly commented and documented source, and the internals-oriented discussion, at the previously mentioned "Python Cookbook" URL.

User Interaction Control

An important requirement is to be able to define interactions with the user in much the same way that you define interactions between software components. You have to be more careful with users, of course, since their behaviors are not programmed, and you can therefore expect them to provide dangerous operating systems commands as first and last names simply in the hope of seeing what they can provoke. If you examined the `pyforms` form handling module and the `pySQL` module for the Xitami example earlier in this chapter, you are now aware of a number of data-driven approaches to forms handling (and of some of the attendant risks).

If a forms handling machine is sufficiently flexible, then all it really needs is an adequate description of the form contents and a knowledge of its place in a workflow that specifies how each form is processed and what interactions take place with databases along the way. The work I have described in this area should have led you to expect a movement toward storing forms descriptions as relational data. To date, I have not found an existing package that does this. Others have developed libraries of varying degrees of sophistication, but nothing that would persuade me to leave the homegrown system I find so easy to use. Perhaps I will be able to incorporate database-driven forms handling in later versions.

One technology I *am* keeping an eye on, though, is *XML Forms for Webware*, a package under construction by Paul Boddie, whose name you read at the start of the chapter as the compiler of a web technology list. The package can be downloaded from `http://thor.prohosting.com/~pboddie/`. The documentation is much more detailed than most treatments of user interaction and addresses issues of state management as well as other development and maintenance techniques. Because Webware servlets are persistent, a servlet can include a description of the form it is processing as a part of its state. This makes it simpler to repeat the form with selective error messages if some inputs do not meet verification or validation criteria. That Paul chose Webware as his platform underlines the architecture's flexibility.

Component Interaction: Choose Your Model

Finally, you should consider how the various components of your web framework are going to interact. Some frameworks, such as Webware and Xitami, include facilities for interaction between their components. You can always use socket programming for such interactions, but this has two potential disadvantages. The first is simply the amount of programming required, and the second is the lack of generality.

If your components need to communicate with others, there are two systems in general use that are designed to facilitate such interactions, and another two newer ones that focus on the use of a combination of XML and HTTP. Because they are all complex, it is only possible to give the briefest of outline to close this chapter.

The Common Object Request Broker Architecture (CORBA)

CORBA is a heterogeneous system for locating and using services in a networked environment. It is an open standard maintained by the Object Management Group (OMG), whose web site (`http://www.omg.org`) prominently features CORBA. The site also describes UML, the Unified Modeling Language used to describe services, and CWM, the Common Warehouse MetaModel, which describes complex data structures for interchange between structured repositories. These last two topics are not considered here.

You describe a CORBA service interface in the OMG Interface Definition Language (IDL). This allows you to specify the data to be passed between objects in a platform-independent manner. Python CORBA interfaces can parse IDL to allow CORBA-mediated interactions between arbitrary system components using the features of Xerox PARC's ILU project, for which Python provides the reference implementation. For a full description see `ftp://ftp.parc.xerox.com/pub/ilu/ilu.html`. ILU is a free, CORBA-compliant, object request broker (ORB) that supplies distributed object connectivity to myriad platforms using a variety of programming languages.

The CORBA architecture is designed to allow clients and servers of particular services to locate each other on the Net and interact by means of what CORBA calls transactions. These are simply programmed interactions rather than the database transactions discussed in earlier chapters.

Microsoft's Distributed Common Object Model (DCOM)

Mark Hammond, Greg Stein, and others have interfaced Python to Microsoft's COM, Distributed COM, and ActiveX architectures. This means, among other things, that you can use Python in ASP or as a COM controller (for example, to automatically extract from or insert information into Excel, Access, or any other COM-aware application). Python can even be an Active Scripting host (which means you could embed JScript inside a Python application, if you had a strange sense of humor).

Python support for many Microsoft Foundation Classes (MFC), COM, DCOM, and Active Scripting is integrated into the ActivePython distribution available from `www.activestate.com`. You can also download the win32all extension package separately if you already have Python installed on your Windows computer.

The major disadvantage of DCOM when compared with CORBA is its restriction to Microsoft-supported environments. DCOM is not an open system because Microsoft controls the standards. This might not matter if you work exclusively with Windows anyway. There has been some work to bridge the CORBA and DCOM worlds using IIOP, the Internet Inter-ORB Protocol, but this has not so far led to massive integration between the Microsoft world and everything else.

DCOM has also proved to be difficult to scale to larger systems without explicit use of MTS. Although this is an acceptable overhead for large-scale projects, it appears to require disproportionate effort for small- and medium-sized developments. A good description of DCOM technology, along with many examples of how Python can be integrated into the DCOM environment, is available in *Programming Python on Win32* (Mark Hammond and Andy Robinson, O'Reilly).

XML-Based Interactions

Various communication schemes are current that use XML to encode remote requests and responses between system components, the two best known being XML-RPC and SOAP. *SOAP* is of interest to web developers because it uses HTTP as the request/response transmission protocol and so is available wherever web services can operate. Microsoft has built its BizTalk services around SOAP, carving out an early market share in distributed services. The XML-RPC camp promotes technologies that allow a similar style of interactions, with all data interchanged using XML.

It will be interesting to see whether BizTalk is operated as a closed framework: if third parties do not run BizTalk servers, then even Microsoft may not be able to provide a large enough infrastructure to win out over an open systems protocol. SOAP itself is receiving broad support across the industry, by companies as large as IBM, so its future seems assured independently of BizTalk.

One of the "features" used to promote SOAP and similar techniques is that it uses HTTP, which corporate firewalls already accommodate. What this really means, however, is that any server that can be accessed for web services can also be accessed for remote procedure calls after it becomes SOAP-capable. This would seem to be a somewhat specious argument in favor of SOAP because opening firewalls to other RPC protocols like IIOP would only involve substantially the same risks. Whether the managers of corporate firewalls will want to allow SOAP traffic to continue to pass unhindered after they appreciate its implications remains to be seen.

Summary

Python can play a role in most areas of the web, as can be clearly seen from a review of existing and developing technologies. The availability of this object-oriented, high-level scripting language has the potential to allow the experimental development of new and different web architectures that can be successfully deployed in production environments on the Internet.