Mod_python Manual

Release 2.7.2

Gregory Trubetskoy

February 10, 2001

E-mail: grisha@modpython.org

This software consists of an extension to the Apache http server. More information about Apache may be found at

http://www.apache.org/

More information on Python language can be found at

http://www.python.org/

Abstract

Mod_python allows embedding Python within the Apache server for a considerable boost in performance and added flexibility in designing web based applications.

This document aims to be the only necessary and authoritative source of information about mod_python, usable as a comprehensive refence, a user guide and a tutorial all-in-one.

See Also:

Python Language Web Site
(http://www.python.org/)
for information on the Python language

Apache Server Web Site
(http://httpd.apache.org/)
for information on the Apache server

CONTENTS

CHAPTER

ONE

Introduction

1.1 Performance

Some very quick tests showed a very apparent performance increase:

```
Platform:       300Mhz Pentium MMX (Sony Vaio PCG-505TR), FreeBSD
Program:        A script that first imported the standard library
                cgi module, then output a single word "Hello!".
Measuring tool: ab (included with apache), 1000 requests.

Standard CGI:   5 requests/s
Cgihandler:     40 requests/s
As a handler:   140 requests/s
```

1.2 Flexibility

Apache processes requests in phases (e.g. read the request, parse headers, check access, etc.). These phases can be implemented by functions called handlers. Traditionally, handlers are written in C and compiled into Apache modules. Mod_python provides a way to extend Apache functionality by writing Apache handlers in Python. For a detailed description of the Apache request processing process, see the *Apache API Notes*.

For most programmers, the request and the authentication handlers provide everything required.

To ease migration from CGI and Httpdapy, two handlers are provided that simulate these environments allowing a user to run his scripts under mod_python with (for the most part) no changes to the code.

See Also:

http://dev.apache.org/
Apache Developer Resources

1.3 History

Mod_python originates from a project called Httpdapy. For a long time Httpdapy was not called mod_python because Httpdapy was not meant to be Apache-specific. Httpdapy was designed to be cross-platform and in fact was initially written for the Netscape server.

This excerpt from the Httpdapy README file describes well the challenges and the solution provided by embedding Python within the HTTP server:

> While developing my first WWW applications a few years back, I found
> that using CGI for programs that need to connect to relational
> databases (commercial or not) is too slow because every hit requires
> loading of the interpreter executable which can be megabytes in size,
> any database libraries that can themselves be pretty big, plus, the
> database connection/authentication process carries a very significant
> overhead because it involves things like DNS resolutions, encryption,
> memory allocation, etc.. Under pressure to speed up the application, I
> nearly gave up the idea of using Python for the project and started
> researching other tools that claimed to specialize in www database
> integration. I did not have any faith in MS's ASP; was quite
> frustrated by Netscape LiveWire's slow performance and bugginess; Cold
> Fusion seemed promising, but I soon learned that writing in html-like
> tags makes programs as readable as assembly. Same is true for
> PHP. Besides, I *really* wanted to write things in Python.
>
> Around the same time the Internet Programming With Python book came
> out and the chapter describing how to embed Python within Netscape
> server immediately caught my attention. I used the example in my
> project, and developed an improved version of what I later called
> Nsapy that compiled on both Windows NT and Solaris.
>
> Although Nsapy only worked with Netscape servers, it was a very
> intelligent generic OO design that, in the spirit of Python, that lent
> itself for easy portability to other web servers.
>
> Incidently, the popularity of Netscape's servers was taking a turn
> south, and so I set out to port Nsapy to other servers starting with
> the most popular one, Apache. And so from Nsapy was born Httpdapy.

...continuing this saga, I later learned that writing Httpdapy for every server is a task a little bigger and less interesting than I originally imagined.

Instead, it seemed like providing a Python counterpart to the popular Perl Apache extension mod_perl that would give Python users the same (or better) capability would be a much more exciting thing to do.

And so it was done.

CHAPTER

TWO

Installation

NOTE: By far the best place to get help with installation and other issues is the mod_python mailing list. Please take a moment to join the mod_python mailing list by sending an e-mail with the word ”subscribe” in the subject to mod_python-request@modpython.org.

2.1 Prerequisites

- Python 1.5.2, 1.6 or 2.0
- Apache 1.3 (1.3.12 or higher recommended, 2.0 is not yet supported)

You will need to have the include files for both Apache and Python, as well as the Python library installed on your system. If you installed Python and Apache from source, then you already have everything that’s needed. However, if you are using prepackaged software (e.g. Linux Red Hat RPM, Debian, or Solaris packages from sunsite, etc) then chances are, you just have the binaries and not the sources on your system. Often, the include files and the libraries are part of separate ”development” package. If you are not sure whether you have all the necessary files, either compile and install Python and Apache from source, or refer to the documentation for your system on how to get the development packages.

2.2 Compiling

There are two ways that this module can be compiled and linked to Apache - statically, or as a DSO (Dynamic Shared Object).

Static linking is a more ”traditional” approach, and most programmers prefer it for its simplicity. The drawback is that it entails recompiling Apache, which some people cannot do for a variety of reasons.

DSO is a newer and still somewhat experimental approach. The module gets compiled as a library that is dynamically loaded by the server at run time. A more detailed description of the Apache DSO mechanism is available at http://www.apache.org/docs/dso.html.

The advantage of DSO is that a module can be installed without recompiling Apache and used as needed. DSO has its disadvantages, however. Compiling a module like mod_python into a DSO can be a complicated process because Python, depending on configuration, may rely on a number of other libraries, and you need to make sure that the DSO is statically linked against each of them. Luckily, the configure script below will spare you of this headache by automatically figuring out all the necessary parameters.

2.2.1 Running ./configure

The **./configure** script will analyze your environment and create custom Makefiles particular to your system. Aside from all the standard autoconf stuff, **./configure** does the following:

- Finds out whether a program called **apxs** is available. This program is part of the standard Apache distribution, and is necessary for DSO compilation. If apxs cannot be found in your $PATH or in /usr/local/apache/bin, DSO compilation will not be available.

  You can manually specify the location of apxs by using the **--with-apxs** option, e.g.:

  ```
  $ ./configure --with-apxs=/usr/local/apache/bin/apxs
  ```

- Checks for **--with-apache** option. Use this option to tell **./configure** where the Apache sources are on your system. The Apache sources are necessary for static compilation. If you do not specify this option, static compilation will not be available. Here is an example:

  ```
  $ ./configure --with-apache=../src/apache_1.3.12 --with-apxs=/usr/local/apache/bin/apxs
  ```

- Checks your Python version and attempts to figure out where **libpython** is by looking at various parameters compiled into your Python binary. By default, it will use the **python** program found in your $PATH.

  If the Python installation on your system is not suitable for mod_python (which can be the case if Python is compiled with thread support), you can specify an alternative location with the **--with-python** options. This option needs to point to the root directory of the Python source, e.g.:

  ```
  $ ./configure --with-python=/usr/local/src/Python-2.0
  ```

  Note that the directory needs to contain already configured and compiled Python. In other words, you must at least run **./configure** and **make**.

2.2.2 Running make

- If possible, the **./configure** script will default to DSO compilation, otherwise, it will default to static. To stay with whatever **./configure** decided, simply run

  ```
  $ make
  ```

  Or, if you would like to be specific, give **make** a **dso** or **static** target:

  ```
  $ make dso
  ```

  OR

  ```
  $ make static
  ```

2.3 Installing

2.3.1 Running make install

- This part of the installation needs to be done as root.

```
$ su
# make install
```

  - For DSO, this will simply copy the library into your Apache libexec directory, where all the other modules are.
  - For static, it will copy some files into your Apache source tree.
  - Lastly, it will install the Python libraries in site-packages and compile them.

  NB: If you wish to selectively install just the Python libraries, the static library or the DSO (which may not always require superuser privileges), you can use the following **make** targets: **install_py_lib**, **install_static** and **install_dso**

2.3.2 Configuring Apache

- If you compiled mod_python as a DSO, you will need to tell Apache to load the module by adding the following line in the Apache configuration file, usually called httpd.conf or apache.conf:

```
LoadModule python_module libexec/mod_python.so
```

  The actual path to **mod_python.so** may vary, but make install should report at the very end exactly where **mod_python.so** was placed and how the `LoadModule` directive should appear.

  If your Apache configuration uses `ClearModuleList` directive, you will need to add mod_python to the module list in the Apache configuration file:

```
AddModule mod_python.c
```

  NB: Some (not all) RedHat Linux users reported that mod_python needs to be first in the module list, or Apache will crash.

- If you used the static installation, you now need to recompile Apache:

```
$ cd ../src/apache_1.3.12
$ ./configure --activate-module=src/modules/python/libpython.a
$ make
```

  Or, if you prefer the old "Configure" style, edit src/Configuration to have

```
AddModule modules/python/libpython.a
```

  then run

```
$ cd src
$ ./Configure
$ Make
```

2.4 Testing

1. Make some directory that would be visible on your web site, for example, htdocs/test.
2. Add the following Apache directives, which can appear in either the main server configuration file, or .htaccess. If you are going to be using the .htaccess file, you will not need the `<Directory>` tag below, and you will need to make sure the `AllowOverride` directive applicable to this directory has at least `FileInfo` specified. (The default is `None`, which will not work.)

   ```
   <Directory /some/directory/htdocs/test>
     AddHandler python-program .py
     PythonHandler mptest
     PythonDebug On
   </Directory>
   ```

   (Substitute /some/directory above for something applicable to your system, usually your Apache ServerRoot)
3. At this time, if you made changes to the main configuration file, you will need to restart Apache in order for the changes to take effect.
4. Edit mptest.py file in the htdocs/test directory so that is has the following lines (be careful when cutting and pasting from your browser, you may end up with incorrect indentation and a syntax error):

   ```
   from mod_python import apache

   def handler(req):
       req.send_http_header()
       req.write("Hello World!")
       return apache.OK
   ```

5. Point your browser to the URL referring to the mptest.py; you should see `"Hello World!"`. If you didn't - refer to the troubleshooting section next.
6. If everything worked well, move on to Chapter 3, *Tutorial*.

2.5 Troubleshooting

There are a couple things you can try to identify the problem:

- Carefully study the error output, if any.
- Check the server error log file, it may contain useful clues.
- Try running Apache from the command line with an -X argument:

  ```
  ./httpd -X
  ```

  This prevents it from backgrounding itself and may provide some useful information.

- Ask on the mod_python list. Make sure to provide specifics such as:
  - Your operating system type, name and version.
  - Your Python version, and any unusual compilation options.
  - Your Apache version.
  - Relevant parts of the Apache config, .htaccess.
  - Relevant parts of the Python code.

CHAPTER

THREE

Tutorial

So how can I make this work?

This is a quick guide to getting started with mod_python programming once you have it installed. This is ***not*** *an installation manual!*

It is also highly recommended to read (at least the top part of) Section 4, Python API *after completing this tutorial.*

3.1 Quick Overview of how Apache Handles Requests

It may seem like a little too much for starters, but you need to understand what a handler is in order to use mod_python. And it's really rather simple.

Apache processes requests in *phases*. For example, the first phase may be to authenticate the user, the next phase to verify whether that user is allowed to see a particular file, then (next phase) read the file and send it to the client. Most requests consist of two phases: (1) read the file and send it to the client, then (2) log the request. Exactly which phases are processed and how varies greatly and depends on the configuration.

A handler is a function that processes one phase. There may be more than one handler available to process a particular phase, in which case they are called in sequence. For each of the phases, there is a default Apache handler (most of which by default perform only very basic functions or do nothing), and then there are additional handlers provided by Apache modules, such as mod_python.

Mod_python provides every possible handler to Apache. Mod_python handlers by default do not perform any function, unless specifically told so by a configuration directive. These directives begin with '`Python`' and end with '`Handler`' (e.g. `PythonAuthenHandler`) and associate a phase with a Python function. So the main function of mod_python is to act as a dispatcher between Apache handlers and Python functions written by a developer like you.

The most commonly used handler is `PythonHandler`. It handles the phase of the request during which the actual content is provided. We will refer to this handler from here on as *generic* handler. The default Apache action for this handler would be to read the file and send it to the client. Most applications you will write will use this one handler. If you insist on seeing all the possible handlers, refer to Section 5, *Apache Directives*.

3.2 So what Exactly does Mod-python do?

Let's pretend we have the following configuration:

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

NB: /mywebdir is an absolute physical path.

And let's say that we have a python program (windows users: substitute forward slashes for backslashes) '/mywedir/myscript.py' that looks like this:

```
from mod_python import apache

def handler(req):

    req.content_type = "text/plain"
    req.send_http_header()
    req.write("Hello World!")

    return apache.OK
```

Here is what's going to happen: The `AddHandler` directive tells Apache that any request for any file ending with '.py' in the '/mywebdir' directory or a subdirectory thereof needs to be processed by mod_python.

When such a request comes in, Apache starts stepping through its request processing phases calling handlers in mod_python. The mod_python handlers check if a directive for that handler was specified in the configuration. (Remember, it acts as a dispatcher.) In our example, no action will be taken by mod_python for all handlers except for the generic handler. When we get to the generic handler, mod_python will notice '`PythonHandler myscript`' directive and do the following:

1. If not already done, prepend the directory in which the `PythonHandler` directive was found to `sys.path`.
2. Attempt to import a module by name `myscript`. (Note that if `myscript` was in a subdirectory of the directory where `PythonHandler` was specified, then the import would not work because said subdirectory would not be in the `sys.path`. One way around this is to use package notation, e.g. '`PythonHandler subdir.myscript`'.)
3. Look for a function called `handler` in `myscript`.
4. Call the function, passing it a `Request` object. (More on what a `Request` object is later)
5. At this point we're inside the script:

    - `from mod_python import apache`

      This imports the apache module which provides us the interface to Apache. With a few rare exceptions, every mod_python program will have this line.

    - `def handler(req):`

      This is our *handler* function declaration. It is called `"handler"` because mod_python takes the name of the directive, converts it to lower case and removes the word `"python"`. Thus `"PythonHandler"` becomes `"handler"`. You could name it something else, and specify it explicitly in the directive using the special '`::`' notation. For example, if the handler function was called '`spam`', then the directive would be '`PythonHandler myscript::spam`'.

Note that a handler must take one argument - the mysterious `Request` object. There is really no mystery about it though. The `Request` object is an object that provides all of the information about this particular request - such as the IP of client, the headers, the URI, etc. The communication back to the client is also done via the `Request` object, i.e. there is no ”response” object.

- ```
  req.content_type = "text/plain"
  ```

  This sets the content type to `"text/plain"`. The default is usually `"text/html"`, but since our handler doesn't produce any html, `"text/plain"` is more appropriate.

- ```
  req.send_http_header()
  ```

  This function sends the HTTP headers to the client. You can't really start writing to the client without sending the headers first. Note that one of the headers is `"Content-Type"`. So if you want to set custom content-types, you better do it before you call `req.send_http_header()`.

- ```
  req.write("Hello Wordl!")
  ```

  This writes the `"Hello World!"` string to the client. (Did I really have to explain this one?)

- ```
  return apache.OK
  ```

  This tells Apache that everything went OK and that the request has been processed. If things did not go OK, that line could be return `apache.HTTP_INTERNAL_SERVER_ERROR` or return `apache.HTTP_FORBIDDEN`. When things do not go OK, Apache will log the error and generate an error message for the client.

Some food for thought: If you were paying attention, you noticed that nowhere did it say that in order for all of the above to happen, the URL needs to refer to myscript.py. The only requirement was that it refers to a .py file. In fact the name of the file doesn't matter, and the file referred to in the URL doesn't have to exist. So, given the above configuration, '`http://myserver/mywebdir/myscript.py`' and '`http://myserver/mywebdir/montypython.py`' would give the exact same result.

At this point, if you didn't understand the above paragraph, go back and read it again, until you do.

3.3 Now something More Complicated - Authentication

Now that you know how to write a primitive handler, let's try something more complicated.

Let's say we want to password-protect this directory. We want the login to be ”spam”, and the password to be ”eggs”.

First, we need to tell Apache to call our *authentication* handler when authentication is needed. We do this by adding the `PythonAuthenHandler`. So now our config looks like this:

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

Notice that the same script is specified for two different handlers. This is fine, because if you remember, mod_python will look for different functions within that script for the different handlers.

Next, we need to tell Apache that we are using Basic HTTP authentication, and only valid users are allowed (this is fairly basic Apache stuff, so I'm not going to go into details here). Our config looks like this now:

```
<Directory /mywebdir>
    AddHandler python-program .py
    PythonHandler myscript
    PythonAuthenHandler myscript
    AuthType Basic
    AuthName "Restricted Area"
    require valid-user
</Directory>
```

Now we need to write an authentication handler function in 'myscript.py'. A basic authentication handler would look like this:

```
def authenhandler(req):

    pw = req.get_basic_auth_pw()
    user = req.connection.user
    if user == "spam" and pw == "eggs":
        return apache.OK
    else:
        return apache.HTTP_UNAUTHORIZED
```

Let's look at this line by line:

- `def authenhandler(req):`

  This is the handler function declaration. This one is called `authenhandler` because, as we already described above, mod_python takes the name of the directive (`PythonAuthenHandler`), drops the word "Python" and converts it lower case.

- ```
  pw = req.get_basic_auth_pw()
  ```

  This is how we obtain the password. The basic HTTP authentication transmits the password in base64 encoded form to make it a little bit less obvious. This function decodes the password and returns it as a string.

- ```
  user = req.connection.user
  ```

  This is how you obtain the username that the user entered. In case you're wondering, the `connection` member of the `Request` object is an object that contains information specific to a *connection*. With HTTP Keep-Alive, a single connection can serve multiple requests.

  NOTE: The two lines above MUST be in that order. The reason is that `connection.user` is assigned a value by the `get_basic_auth_pw()` function. If you try to use the `connection.user` value without calling `get_basic_auth_pw()` first, it will be `None`.

- ```
  if user == "spam" and pw == "eggs":
      return apache.OK
  ```
  ```

We compare the values provided by the user, and if they are what we were expecting, we tell Apache to go ahead and proceed by returning `apache.OK`. Apache will then proceed to the next handler. (which in this case would be `handler()` if it's a `.py` file).

- ```
  else:
      return apache.HTTP_UNAUTHORIZED
  ```

  Else, we tell Apache to return `HTTP_UNAUTHORIZED` to the client.

3.4 Publisher Handler Makes it Easy

At this point you may be wondering if mod_python is all that useful after all. You may find yourself asking: "If there can only be one handler per directory, how am I to structure my application?"

Enter the `publisher` handler provided as one of the standard mod_python handlers. To get the publisher handler working, you will need the following lines in your config:

```
AddHandler python-program .py
PythonHandler mod_python.publisher
```

The following example will demonstrate a simple feedback form. The form will ask for the name, e-mail address and a comment and the will construct an e-mail to the webmaster using the information submitted by the user.

Here is the html for the form:

```
<html>
  Please provide feedback below:
  <p>
  <form action="form/email" method="POST">

    Name: <input type="text" name="name"><br>
    Email: <input type="text" name="email"><br>
    Comment: <textarea name="comment" rows=4 cols=20></textarea><br>
    <input type="submit">

  </form>
</html>
```

Note the `action` element of the `<form>` tag points to `form/email`. We are going to create a file called form.py, like this:
```

```
import smtplib

def email(req, name, email, comment):

    # see if the user provided all the parameters
    if not (name and email and comment):
        return "A required parameter is missing, \
please go back and correct the error"

    # create the message text
    msg = """\
From: %s
Subject: feedback
To: webmaster

I have the following comment:

%s

Thank You,

%s

""" % (email, comment, name)

    # send it out
    conn = smtplib.SMTP("localhost")
    conn.sendmail(email, ["webmaster"], msg)
    conn.quit()

    # provide feedback to the user
    s = """\
<html>

Dear %s,<br>

Thank You for your kind comments, we
will get back to you shortly.

</html>""" % name

    return s
```

When the user clicks the Submit button, the publisher handler will load the `email` function in the `form` module, passing it the form fields as keyword arguments. Note that it will also pass the `Request` object as `req`. Note also that you do not have to have `req` as one of the arguments if you do not need it. The publisher handler is smart enough to pass your function only those arguments that it will accept.

Also notice how it sends data back to the customer - via the return value of the function.

And last, but not the least, note how all the power of mod_python is still available to this function, since it has access to the `Request` object. You can do all the same things you can do with a "native" mod_python handler, e.g. set custom headers via `req.headers_out`, return errors by raising `apache.SERVER_ERROR` exceptions, write or read directly to and from the client via `req.write` and `req.read`, etc.

Read Section 6.1 *Publisher Handler* for more information on the publisher handler.

CHAPTER

FOUR

Python API

4.1 Multiple Interpreters

When working with mod_python, it is important to be aware of a feature of Python that is normally not used when using the language for writing scripts to be run from command line. This feature is not available from within Python itself and can only be accessed through the *C language API*.

Python C API provides the ability to create *subinterpreters*. A more detailed description of a subinterpreter is given in the documentation for the `Py_NewInterpreter()` function. For this discussion, it will suffice to say that each subinterpreter has its own separate namespace, not accessible from other subinterpreters. Subinterpreters are very useful to make sure that separate programs running under the same Apache server do not interfere with one another..

At server start-up or mod_python initialization time, mod_python initializes an interpreter called *main* interpreter. The main interpreter contains a dictionary of subinterpreters. Initially, this dictionary is empty. With every hit, as needed, subinterpreters are created, and references to them are stored in this dictionary. The dictionary is keyed on a string, also known as *interpreter name*. This name can be any string. The main interpreter is named '`main_interpreter`'. The way all other interpreters are named can be controlled by `PythonInterp*` directives. Default behaviour is to name interpreters using the Apache virtual server name (`ServerName` directive). This means that all scripts in the same vrtual server execute in the same subinterpreter, but scripts in different virtual servers execute in different subinterpreters with completely separate namespaces. `PythonInterpPerDirectory` and `PythonInterpPerDirective` directives alter the naming convention to use the absolute path of the directory being accessed, or the directory in which the `Python*Handler` was encountered, respectively.

Once created, a subinterpreter will be reused for subsequent requests. It is never destroyed and exists until the Apache child process dies.

See Also:

Python C Language API
(http://www.python.org/doc/current/api/api.html)
Python C Language API

4.2 Overview of a Handler

A *handler* is a function that processes a particular phase of a request. Apache processes requests in phases - read the request, process headers, provide content, etc. For every phase, it will call handlers, provided by either the Apache core or one of its modules, such as mod_python, which passes control to functions provided b the user and written in Python. A handler written in Python is not any different than a handler written in C, and follows these rules:

A handler function will always be passed a reference to a `Request` object. (Throughout this manual, the `Request` object is often referred to by the `req` variable.)

Every handler can return:

- `apache.OK`, meaning this phase of the request was handled by this handler and no errors occurred.
- `apache.DECLINED`, meaning this handler refused to handle this phase of the requestand Apache needs to look for another handler.
- `apache.`*`HTTP_ERROR`*, meaning an HTTP error occurred. *HTTP_ERROR* can be:

```
HTTP_CONTINUE                    = 100
HTTP_SWITCHING_PROTOCOLS         = 101
HTTP_PROCESSING                  = 102
HTTP_OK                          = 200
HTTP_CREATED                     = 201
HTTP_ACCEPTED                    = 202
HTTP_NON_AUTHORITATIVE           = 203
HTTP_NO_CONTENT                  = 204
HTTP_RESET_CONTENT               = 205
HTTP_PARTIAL_CONTENT             = 206
HTTP_MULTI_STATUS                = 207
HTTP_MULTIPLE_CHOICES            = 300
HTTP_MOVED_PERMANENTLY           = 301
HTTP_MOVED_TEMPORARILY           = 302
HTTP_SEE_OTHER                   = 303
HTTP_NOT_MODIFIED                = 304
HTTP_USE_PROXY                   = 305
HTTP_TEMPORARY_REDIRECT          = 307
HTTP_BAD_REQUEST                 = 400
HTTP_UNAUTHORIZED                = 401
HTTP_PAYMENT_REQUIRED            = 402
HTTP_FORBIDDEN                   = 403
HTTP_NOT_FOUND                   = 404
HTTP_METHOD_NOT_ALLOWED          = 405
HTTP_NOT_ACCEPTABLE              = 406
HTTP_PROXY_AUTHENTICATION_REQUIRED= 407
HTTP_REQUEST_TIME_OUT            = 408
HTTP_CONFLICT                    = 409
HTTP_GONE                        = 410
HTTP_LENGTH_REQUIRED             = 411
HTTP_PRECONDITION_FAILED         = 412
HTTP_REQUEST_ENTITY_TOO_LARGE    = 413
HTTP_REQUEST_URI_TOO_LARGE       = 414
HTTP_UNSUPPORTED_MEDIA_TYPE      = 415
HTTP_RANGE_NOT_SATISFIABLE       = 416
HTTP_EXPECTATION_FAILED          = 417
HTTP_UNPROCESSABLE_ENTITY        = 422
HTTP_LOCKED                      = 423
HTTP_FAILED_DEPENDENCY           = 424
HTTP_INTERNAL_SERVER_ERROR       = 500
HTTP_NOT_IMPLEMENTED             = 501
HTTP_BAD_GATEWAY                 = 502
HTTP_SERVICE_UNAVAILABLE         = 503
HTTP_GATEWAY_TIME_OUT            = 504
HTTP_VERSION_NOT_SUPPORTED       = 505
HTTP_VARIANT_ALSO_VARIES         = 506
HTTP_INSUFFICIENT_STORAGE        = 507
HTTP_NOT_EXTENDED                = 510
```

As an alternative to *returning* an HTTP error code, handlers can signal an error by *raising* the `apache.SERVER_RETURN` exception, and providing an HTTP error code as the exception value, e.g.

```
raise apache.SERVER_RETURN, apache.HTTP_FORBIDDEN
```

Handlers can send content to the client using the `Request.write()` method. Before sending the body of the response, headers must be sent using the `Request.send_http_header()` method.

Client data, such as POST requests, can be read by using the `Request.read()` function.

NOTE: The directory of the Apache `Python*Handler` directive in effect is prepended to the `sys.path`. If the directive was specified in a server config file outside any `<Directory>`, then the directory is unknown and not prepended.

An example of a minimalistic handler might be:

```
from mod_python import apache

def requesthandler(req):
    req.content_type = "text/plain"
    req.send_http_header()
    req.write("Hello World!")
    return apache.OK
```

4.3 apache – Access to Apache Internals.

The Python Application Programmer interface to Apache internals is contained in a module appropriately named `apache`, located inside the `mod_python` package. This module provides some important objects that map to Apache internal structures, as well as some useful functions, all documented below.

The `apache` module can only be imported by a script running under mod_python. This is because it depends on a built-in module `_apache` provided by mod_python. It is best imported like this:

```
from mod_python import apache
```

`mod_python.apache` module defines the following objects and functions. For a more in-depth look at Apache internals, see the *Shambhala API Notes*

`log_error`(*message*[, *level, server*])

An interface to the Apache *ap_log_error()* function. *message* is a string with the error message, *level* is one of the following constants:

```
APLOG_EMERG
APLOG_ALERT
APLOG_CRIT
APLOG_ERR
APLOG_WARNING
APLOG_NOTICE
APLOG_INFO
APLOG_DEBUG
APLOG_NOERRNO
```

server is a reference to a `Request.server` object. If *server* is not specified, then the error will be logged to

the default error log, otherwise it will be written to the error log for the appropriate virtual server.

make_table()
Returns a new empty object of type `mp_table`. See Section 4.3.1 for a description of a table object.

4.3.1 Table Object (mp_table)

The table object is a Python mapping to the Apache table. The table object performs just like a dictionary, with the only difference that key lookups are case insensitive.

Much of the information that Apache uses is stored in tables. For example, `Request.header_in` and `Request.headers_out`.

All the tables that mod_python provides inside the `Request` object are actual mappings to the Apache structures, so changing the Python table also changes the underlying Apache table.

In addition to normal dictionary-like behavior, the table object also has the following method:

add(*key, val*)
`add()` allows for creating duplicate keys, which is useful when multiple headers, such as `Set-Cookie:` are required.

4.3.2 Request Object

The request object is a Python mapping to the Apache `request_rec` structure.

The request object is a real Python object. You can dynamically assign attributes to it as a way to communicate between handlers.

When a handler is invoked, it is always passed a single argument - the `Request` object.

Request Methods

add_handler(*htype, handler*[, *dir*])
Allows dynamic handler registration. *htype* is a string containing the name of any of the apache `Python*Handler` directives, e.g. '`PythonHandler`'. *handler* is a string containing the name of the module and the handler function. Optional *dir* is a string containing the name of the directory to be added to the pythonpath. If no directory is specified, then, if there is already a handler of the same type specified, its directory is inherited, otherwise the directory of the presently executing handler is used.

A handler added this way only persists throughout the life of the request. It is possible to register more handlers while inside the handler of the same type. One has to be careful as to not to create an infinite loop this way.

Dynamic handler registration is a useful technique that allows the code to dynamically decide what will happen next. A typical example might be a `PythonAuthenHandler` that will assign different `PythonHandlers` based on the authorization level, something like:

```
if manager:
    req.add_handler("PythonHandler", "menu::admin")
else:
    req.add_handler("PythonHandler", "menu::basic")
```

Note: There is no checking being done on the validity of the handler name. If you pass this function an invalid handler it will simply be ignored.

add_common_vars()
Calls the Apache `ap_add_common_vars()` function. After a call to this method, `Request.subprocess_env` will contain a lot of CGI information.

child_terminate()
Terminate a child process. This should terminate the current child process in a nice fashion.

This method does nothing in multithreaded environments (e.g. Windows).

get_basic_auth_pw()
Returns a string containing the password when Basic authentication is used.

get_config()
Returns a reference to the table object containing the configuration in effect for this request. The table has directives as keys, and their values, if any, as values.

get_dirs()
Returns a reference to the table object keyed by directives currently in effect and having directory names of where the particular directive was last encountered as values. For every key in the table returned by `get_config()`, there will be a key in this table. If the directive was in one of the server config files outside of any `<Directory>`, then the value will be an empty string.

get_remote_host(*type*)
Returns the a string with the remote client's DNS name or IP or `None` on failure. The first call to this function may entail a DNS look up, but subsequent calls will use the cached result from the first call.

The optional type argument can specify the following:

- `apache.REMOTE_HOST` Look up the DNS name. Fail if Apache directive `HostNameLookups` is `off` or the hostname cannot be determined.
- `apache.REMOTE_NAME` *(Default)* Return the DNS name if possible, or the IP (as a string in dotted decimal notation) otherwise.
- `apache.REMOTE_NOLOOKUP` Don't perform a DNS lookup, return an IP. Note: if a lookup was performed prior to this call, then the cached host name is returned.
- `apache.REMOTE_DOUBLE_REV` Force a double-reverse lookup. On failure, return None.

get_options()
Returns a reference to the table object containing the options set by the `PythonOption` directives.

read([*len*])
Reads at most *len* bytes directly from the client, returning a string with the data read. If the *len* argument is negative or ommitted, reads all data given by the client.

This function is affected by the `Timeout` Apache configuration directive. The read will be aborted and an `IOError` raised if the `Timeout` is reached while reading client data.

This function relies on the client providing the `Content-length` header. Absense of the `Content-length` header will be treated as if `Content-length: 0` was supplied.

Incorrect `Content-length` may cause the function to try to read more data than available, which will make the function block until a `Timeout` is reached.

readline([*len*])
Like `read()` but reads until end of line.

Note that in accordance with the HTTP specification, most clients will be terminating lines with "\r\n" rather than simply "\n".

register_cleanup(*callable*[, *data*])
Registers a cleanup. Argument *callable* can be any callable object, the optional argument *data* can be any object (default is `None`). At the very end of the request, just before the actual request record is destroyed by Apache, *callable* will be called with one argument, *data*.

send_http_header()
Starts the output from the request by sending the HTTP headers. This function has no effect when called more

than once within the same request. Any manipulation of `Request.headers_out` after this function has been called is pointless since the headers have already been sent to the client.

`write`(*string*)
Writes *string* directly to the client, then flushes the buffer.

Request Members

`connection`
A `connection` object associated with this request. See Connection Object below for details. *(Read-Only)*

`server`
A server object associate with this request. See Server Object below for details. *(Read-Only*)

`next`
If this is an internal redirect, the `request` object we redirect to. *(Read-Only*)

`prev`
If this is an internal redirect, the `request` object we redirect from. *(Read-Only*)

`main`
If this is a sub-request, pointer to the main request. *(Read-Only*)

`the_request`
String containing the first line of the request. *(Read-Only*)

`assbackwards`
Is this an HTTP/0.9 "simple" request? *(Read-Only*)

`header_only`
A boolean value indicating HEAD request, as opposed to GET. *(Read-Only*)

`protocol`
Protocol, as given by the client, or "HTTP/0.9". Same as CGI $SERVER_PROTOCOL. *(Read-Only*)

`proto_num`
Integer. Number version of protocol; 1.1 = 1001 *(Read-Only*)

`request_time`
A long integer. When request started. *(Read-Only*)

`status_line`
Status line. E.g. "200 OK". *(Read-Only*)

`method`
A string containing the method - 'GET', 'HEAD', 'POST', etc. Same as CGI $REQUEST_METHOD. *(Read-Only*)

`method_number`
Integer containg the method number. *(Read-Only*)

`allowed`
Integer. A bitvector of the allowed methods. Used in relation with METHOD_NOT_ALLOWED. *(Read-Only*)

`sent_body`
Integer. Byte count in stream is for body. (?) *(Read-Only*)

`bytes_sent`
Long integer. Number of bytes sent. *(Read-Only*)

`mtime`
Long integer. Time the resource was last modified. *(Read-Only*)

boundary
String. Multipart/byteranges boundary. *(Read-Only)*

range
String. The `Range:` header. *(Read-Only)*

clength
Long integer. The "real" content length. (I.e. can only be used after the content's been read?) *(Read-Only)*

remaining
Long integer. Bytes left to read. (Only makes sense inside a read operation.) *(Read-Only)*

read_length
Long integer. Number of bytes read. *(Read-Only)*

read_body
Integer. How the request body should be read. (?) *(Read-Only)*

read_chunked
Boolean. Read chunked transfer coding. *(Read-Only)*

headers_in
A table object containing headers sent by the client.

headers_out
A `table` object representing the headers to be sent to the client. Note that manipulating this table after the `Request.send_http_headers()` has been called is meaningless, since the headers have already gone out to the client.

err_headers_out
These headers get send with the error response, instead of headers_out.

handler
The hame of the handler currently being processed. In all cases with mod_python, this should be "python-program". *(Read-Only)*

content_encoding
String. Content encoding. *(Read-Only)*

vlist_validator
Integer. Variant list validator (if negotiated). *(Read-Only)*

no_cache
Boolean. No cache if true. *(Read-Only)*

no_local_copy
Boolean. No local copy exists. *(Read-Only)*

unparsed_uri
The URL without any parsing performed. *(Read-Only)*

uri
The path portion of the URI. *(Read-Only)*

filename
String. File name being requested. *(Read-Only)*

path_info
String. What follows after the file name, but is before query args, if anything. Same as CGI $PATH_INFO. *(Read-Only)*

args
String. Same as CGI $QUERY_ARGS. *(Read-Only)*

4.3.3 Connection Object (mp_conn)

The connection object is a Python mapping to the Apache conn_rec structure.

Connection Members

`server`
A `server` object associated with this connection. *(Read-Only)*

`base_server`
A `server` object for the physical vhost that this connection came in through. *(Read-Only)*

`child_num`
Integer. The number of the child handling the request. *(Read-Only)*

`local_addr`
The (address, port) tuple for the server. *(Read-Only)*

`remote_addr`
The (address, port) tuple for the client. *(Read-Only)*

`remote_ip`
String with the IP of the client. Same as CGI $REMOTE_ADDR. *(Read-Only)*

`remote_host`
String. The DNS name of the remote client. None if DNS has not been checked, "" (empty string) if no name found. Same as CGI $REMOTE_HOST. *(Read-Only)*

`remote_logname`
Remote name if using RFC1413 (ident). Same as CGI $REMOTE_IDENT. *(Read-Only)*

`user`
If an authentication check is made, this will hold the user name. **NOTE:** You must call `get_basic_auth_pw()` before using this value. Same as CGI $REMOTE_USER. *(Read-Only)*

`ap_auth_type`
Authentication type. (None == basic?). Same as CGI $AUTH_TYPE. *(Read-Only)*

`keepalives`
The number of times this connection has been used. (?) *(Read-Only)*

`local_ip`
String with the IP of the server. *(Read-Only)*

`local_host`
DNS name of the server. *(Read-Only)*

4.3.4 Server Object (mp_server)

The request object is a Python mapping to the Apache `request_rec` structure. The server structure describes the server (possibly virtual server) serving the request.

Server Methods

`register_cleanup`(*request, callable*[, *data*])
Registers a cleanup. Very similar to `req.register_cleanup()`, except this cleanup will be executed at child termination time. This function requires one extra argument - the request object.

Server Members

defn_name
String. The name of the configuration file where the server definition was found. *(Read-Only*)

defn_line_number
Integer. Line number in the config file where the server definition is found. *(Read-Only*)

srm_confname
Location of the srm config file. *(Read-Only*)

server_admin
Value of the `ServerAdmin` directive. *(Read-Only*)

server_hostname
Value of the `ServerName` directive. Same as CGI $SERVER_NAME. *(Read-Only*)

port
Integer. TCP/IP port number. Same as CGI $SERVER_PORT. *(Read-Only*)

error_fname
The name of the error log file for this server, if any. *(Read-Only*)

loglevel
Integer. Logging level. *(Read-Only*)

is_virtual
Boolean. True if this is a virtual server. *(Read-Only*)

timeout
Integer. Value of the `Timeout` directive. *(Read-Only*)

keep_alive_timeout
Integer. Keepalive timeout. *(Read-Only*)

keep_alive_max
Maximum number of requests per keepalive. *(Read-Only*)

send_buffer_size
Integer. Size of the TCP send buffer. *(Read-Only*)

path
String. Path for `ServerPath` *(Read-Only*)

pathlen
Integer. Path length. *(Read-Only*)

server_uid
UID under which the server is running. *(Read-Only*)

server_gid
GID under which the server is running. *(Read-Only*)

4.3.5 Debugging

Mod_python supports the ability to execute handlers within the Python debugger (pdb) via the `PythonEnablePdb` Apache directive. Since the debugger is an interactive tool, httpd must be invoked with the -X option. (NB: When pdb starts, you will not see the usual `>>>` prompt. Just type in the pdb commands like you would if there was one.)

4.3.6 Internal Callback Object

The Apache server interfaces with the Python interpreter via a callback object obCallBack. When a subinterpreter is created, an instance of obCallBack is created in this subinterpreter. Interestingly, obCallBack is not written in C, it is written in Python and the code for it is in the apache module. Mod_python only uses the C API to import apache and then instantiate obCallBack, storing a reference to the instance in the interpreter dictionary described above. Thus, the values in the interpreter dictionary are callback object instances.

When a request handler is invoked by Apache, mod_python uses the obCallBack reference to call its method Dispatch, passing it the name of the handler being invoked as a string.

The Dispatch method then does the rest of the work of importing the user module, resolving the callable object in it and calling it passing it a request object.

4.4 `util` – Miscellaneous Utilities

The `util` module provides a number of utilities handy to a web application developer.

The functionality provided by `util` is also available in the standard Python library `cgi` module, but the implementation in `cgi` is specific to the CGI environment, making it not the most efficient one for mod_python. For example, the code in `util` does not use the environment variables since most of the information is available directly from the `Request` object. Some of the functions in the `util` module are implemented in C for even better performance.

The recommended way of using this module is:

```
from mod_python import util
```

See Also:

Common Gateway Interface RFC Project Page
(http://CGI-Spec.Golux.Com/)
for detailed information on the CGI specification

4.4.1 FieldStorage class

Access to form data is provided via the `FieldStorage` class. This class is similar to the standard library module `cgi` `FieldStorage` (but there are a few differences).

`FieldStorage`(*req*[, *keep_blank_values*, *strict_parsing*])
This class provides uniform access to HTML form data submitted by the client. *req* is an instance of the mod_python `Request` object.

The optional argument *keep_blank_values* is a flag indicating whether blank values in URL encoded form data should be treated as blank strings. The default is false, which means that blank values are ignored as if they were not included.

The optional argument *strict_parsing* is not yet implemented.

While being instantiated, the `FieldStorage` class reads all of the data provided by the client. Since all data provided by the client is consumed at this point, there should be no more than one `FieldStorage` class instantiated per signle request, nor should you make any attempts to read client data before or after instantiating a `FieldStorage`.

The data read from the client is then parsed into separate fields and packaged in `Field` objects, one per field. For HTML form inputs of type `file`, a temporary file is created that can later be accessed via the `file` attribute of a `Field` object.

The `FieldStorage` class has a mapping object interface, i.e. it can be treated like a dictionary. When used as a dictionary, the dictionary keys are form input names, and the returned dictionary value can be:

- A string, containing the form input value. This is only when there is a single value corresponding to the input name.
- An instances of a `Field` class, if the input is a file upload.
- A list of strings and/or `Field` objects. This is when multiple values exist, such as for a `<select>` HTML form element.

Note that unlike the standard library `cgi` module `FieldStorage` class, a `Field` object is returned *only* when it is a file upload. This means that you do not need to use the `.value` attribute to access values of fields in most cases.

In addition to standard mapping object methods, `FieldStorage` objects have the following attributes:

`list`
This is a list of `Field` objects, one for each input. Multiple inputs with the same name will have multiple elements in this list.

Field class

`Field``()`
This class is used internally by `FieldStorage` and is not meant to be instantiated by the user. Each instance of a `Field` class represents an HTML Form input.

`Field` instances have the following attributes:

`name`
The input name.

`value`
The input value. This attribute can be used to read data from a file upload as well, but one has to excercise caution when dealing with large files since when accessed via `value`, the whole file is read into memory.

`file`
This is a file object. For file uploads it points to a temporary file. For simple values, it is a `StringIO` object, so you can read simple string values via this attribute instead of using the `value` attribute as well.

`filename`
The name of the file as provided by the client.

`type`
The content-type for this input as provided by the client.

`type_opyions`
This is what follows the actual content type in the `content-type` header provided by the client, if anything. This is a dictionary.

`disposition`
The value of the first part of the `content-disposition` header.

`disposition_options`
The second part (if any) of the `content-disposition` header in the form of a dictionary.

See Also:

RFC 1867, "*Form-based File Upload in HTML*"
for a description of form-based file uploads

4.4.2 Other functions

parse_qs(*qs*[, *keep_blank_values, strict_parsing*])

This functnion is functionally equivalent to the standard library `cgi parse_qs`, except that it is written in C and is much faster.

Parse a query string given as a string argument (data of type application/x-www-form-urlencoded). Data are returned as a dictionary. The dictionary keys are the unique query variable names and the values are lists of values for each name.

The optional argument *keep_blank_values* is a flag indicating whether blank values in URL encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

Note: The *strict_parsing* argument is not yet implemented.

parse_qsl(*qs*[, *keep_blank_values, strict_parsing*])

This functnion is functionally equivalent to the standard library `cgi parse_qsl`, except that it is written in C and is much faster.

Parse a query string given as a string argument (data of type application/x-www-form-urlencoded). Data are returned as a list of name, value pairs.

The optional argument *keep_blank_values* is a flag indicating whether blank values in URL encoded queries should be treated as blank strings. A true value indicates that blanks should be retained as blank strings. The default false value indicates that blank values are to be ignored and treated as if they were not included.

Note: The *strict_parsing* argument is not yet implemented.

CHAPTER

FIVE

Apache Configuration Directives

5.1 Handlers

5.1.1 Python*Handler Directive Syntax

All **Python*Handler** directives have the following syntax:

`Python*Handler` *`handler [handler] ...`*

Where *handler* is a callable object (e.g. a function) that accepts a single argument - request object.

Multiple handlers can be specified on a single line, in which case they will be called sequentially, from left to right. Same handler directives can be specified multiple times as well, with the same result - all handlers listed will be executed sequentially, from first to last. If any handler in the sequence returns a value other than `apache.OK`, then execution of all subsequent handlers is aborted.

A *handler* has the following syntax:

`module[::object] [module::[object]] ...`

Where module can be a full module name (package dot notation is accepted), and the optional object is the name of an object inside the module.

Object can also contain dots, in which case it will be resolved from left to right. During resolution, if mod_python encounters an object of type `<class>`, it will try instantiate it passing it a single argument, a request object.

If no object is specified, then it will default to the directive of the handler, all lower case, with the word '`Python`' removed. E.g. the default object for PythonAuthenHandler would be authenhandler.

Example:

```
PythonAuthzHandler mypackage.mymodule::checkallowed
```

For more information on handlers, see Overview of a Handler.

Side note: The "::" was chosen for performance reasons. In order for Python to use objects inside modules, the modules first need to be imported. However, if the separator were simply a ".", it would involve a much more complex process of sequentially evaluating every word to determine whether it is a package, module, class etc. Using the (admittedly un-Python-like) "::" takes the time consuming work of figuring out where the module ends and the object inside of it begins away from mod_python resulting in a modest performance gain..

5.1.2 PythonPostReadRequestHandler

Syntax: *Python*Handler Syntax*
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

This routine is called after the request has been read but before any other phases have been processed. This is useful to make decisions based upon the input header fields.

NOTE: At the time when this phase of the request is being processed, the URI has not been translated into a path name, therefore this directive will never be executed by Apache if specified within `<Directory>`, `<Location>`, `<File>` directives or in an '.htaccess' file. The only place this can be specified is the main configuration file, and the code for it will execute in the main interpreter.

5.1.3 PythonTransHandler

Syntax: *Python*Handler Syntax*
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

This routine gives allows for an opportunity to translate the URI into an actual filename, before the server's default rules (Alias directives and the like) are followed.

NOTE: At the time when this phase of the request is being processed, the URI has not been translated into a path name, therefore this directive will never be executed by Apache if specified within `<Directory>`, `<Location>`, `<File>` directives or in an '.htaccess' file. The only place this can be specified is the main configuration file, and the code for it will execute in the main interpreter.

5.1.4 PythonHeaderParserHandler

Syntax: *Python*Handler Syntax*
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

This handler is called to give the module a chance to look at the request headers and take any appropriate specific actions early in the processing sequence.

5.1.5 PythonAccessHandler

Syntax: *Python*Handler Syntax*
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

This routine is called to check for any module-specific restrictions placed upon the requested resource.

For example, this can be used to restrict access by IP number. To do so, you would return `HTTP_FORBIDDEN` or some such to indicate that access is not allowed.

5.1.6 PythonAuthenHandler

Syntax: *Python*Handler Syntax*
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

This routine is called to check the authentication information sent with the request (such as looking up the user in a database and verifying that the [encrypted] password sent matches the one in the database).

To obtain the username, use `req.connection.user`. To obtain the password entered by the user, use the `req.get_basic_auth_pw()` function.

A return of `apache.OK` means the authentication succeeded. A return of `apache.HTTP_UNAUTHORIZED` with most browser will bring up the password dialog box again. A return of `apache.HTTP_FORBIDDEN` will usually show the error on the browser and not bring up the password dialog `again.` `HTTP_FORBIDDEN` should be used when authentication succeeded, but the user is not permitted to access a particular URL.

An example authentication handler might look like this:

```
def authenhandler(req):

    pw = req.get_basic_auth_pw()
    user = req.connection.user
    if user == "spam" and pw == "eggs":
        return apache.OK
    else:
        return apache.HTTP_UNAUTHORIZED
```

Note: `req.get_basic_auth_pw()` must be called prior to using the `req.connection.user` value. Apache makes no attempt to decode the authentication information unless `req.get_basic_auth_pw()` is called.

5.1.7 PythonTypeHandler

Syntax: *Python*Handler Syntax*
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

This routine is called to determine and/or set the various document type information bits, like Content-type (via `r->content_type`), language, et cetera.

5.1.8 PythonFixupHandler

Syntax: *Python*Handler Syntax*
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

This routine is called to perform any module-specific fixing of header fields, et cetera. It is invoked just before any content-handler.

5.1.9 PythonHandler

Syntax: *Python*Handler Syntax*
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

This is the main request handler. 99.99only provide this one handler.

5.1.10 PythonInitHandler

Syntax: *Python*Handler Syntax*
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

This handler is the first handler called in the request processing phases that is allowed both inside and outside '.htaccess' and directory.

This handler is actually an alias to two different handlers. When specified in the main config file outside any directory tags, it is an alias to `PostReadRequestHandler`. When specified inside directory (where `PostReadRequestHandler` is not allowed), it aliases to `PythonHeaderParserHandler`.

(This idea was borrowed from mod_perl)

5.1.11 PythonLogHandler

Syntax: *Python*Handler Syntax*
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

This routine is called to perform any module-specific logging activities over and above the normal server things.

5.1.12 PythonCleanupHandler

Syntax: *Python*Handler Syntax*
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

This is the very last handler, called just before the request object is destroyed by Apache.

Unlike all the other handlers, the return value of this handler is ignored. Any errors will be logged to the error log, but will not be sent to the client, even if PythonDebug is On.

This handler is not a valid argument to the `rec.add_handler()` function. For dynamic clean up registration, use `req.register_cleanup()`.

Once cleanups have started, it is not possible to register more of them. Therefore, `req.register_cleanup()` has no effect within this handler.

Cleanups registered with this directive will execute *after* cleanups registered with `req.register_cleanup()`.

5.2 Other Directives

5.2.1 PythonEnablePdb

Syntax: PythonEnablePdb {On, Off}
Default: PythonEnablePdb Off
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

When On, mod_python will execute the handler functions within the Python debugger pdb using the `pdb.runcall()` function.

Because pdb is an interactive tool, start httpd with the -X option when using this directive.

5.2.2 PythonDebug

Syntax: PythonDebug {On, Off}
Default: PythonDebug Off
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

Normally, the traceback output resulting from uncaught Python errors is sent to the error log. With PythonDebug On directive specified, the output will be sent to the client (as well as the log), except when the error is `IOError` while writing, in which case it will go to the error log.

This directive is very useful during the development process. It is recommended that you do not use it production environment as it may reveal to the client unintended, possibly sensitive security information.

5.2.3 PythonImport

Syntax: PythonImport *module ...*
Context: directory
Module: mod_python.c

Tells the server to import the Python module module at process startup. This is useful for initialization tasks that could be time consuming and should not be done at the request processing time, e.g. initializing a database connection.

The import takes place at child process initialization, so the module will actually be imported once for every child process spawned.

Note that at the time when the import takes place, the configuration is not completely read yet, so all other directives, including PythonInterpreter have no effect on the behavior of modules imported by this directive. Because of this limitation, the use of this directive should be limited to situations where it is absolutely necessary, and the recommended approach to one-time initializations should be to use the Python import mechanism.

The module will be imported within the subinterpreter according with the directory name specified by the `<Directory>` directive. For all other subinterpreters, the module will not appear imported.

See also Multiple Interpreters.

5.2.4 PythonInterpPerDirectory

Syntax: PythonInterpPerDirectory {On, Off}
Default: PythonInterpPerDirectory Off

Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

Instructs mod_python to name subinterpreters using the directory of the file in the request (`req.filename`) rather than the the server name. This means that scripts in different directories will execute in different subinterpreters as opposed to the default policy where scripts in the same virtual server execute in the same subinterpreter, even if they are in different directories.

For example, assume there is a '/directory/subdirectory'. '/directory' has an .htaccess file with a PythonHandler directive. '/directory/subdirectory' doesn't have an .htacess. By default, scripts in /directory and '/directory/subdirectory' would execute in the same interpreter assuming both directories are accessed via the same virtual server. With PythonInterpPerDirectory, there would be two different interpreters, one for each directory.

Note: In early phases of the request prior to the URI translation (PostReadRequestHandler and TransHandler) the path is not yet known because the URI has not been translated. During those phases and with PythonInterpPerDirectory on, all python code gets executed in the main interpreter. This may not be exactly what you want, but unfortunately there is no way around this.

See also Multiple Interpreters.

5.2.5 PythonInterpPerDirective

Syntax: PythonInterpPerDirective {On, Off}
Default: PythonInterpPerDirective Off
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

Instructs mod_python to name subinterpreters using the directory in which the Python*Handler directive currently in effect was encountered.

For example, assume there is a '/directory/subdirectory'. '/directory' has an .htaccess file with a PythonHandler directive. '/directory/subdirectory' has another '.htacess' file with another PythonHandler. By default, scripts in '/directory' and '/directory/subdirectory' would execute in the same interpreter assuming both directories are in the same virtual server. With PythonInterpPerDirective, there would be two different interpreters, one for each directive.

See also Multiple Interpreters.

5.2.6 PythonHandlerModule

Syntax: PythonHandlerModule module
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

PythonHandlerModule can be used an alternative to Python*Handler directives. The module specified in this handler will be searched for existence of functions matching the default handler function names, and if a function is found, it will be executed.

For example, instead of:

```
PythonAutenHandler mymodule
PythonHandler mymodule
PythonLogHandler mymodule
```

one can simply say

```
PythonHandlerModule mymodule
```

5.2.7 PythonNoReload

Syntax: PythonNoReload {On, Off}
Default: PythonNoReload Off
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

Instructs mod_python not to check the modification date of the module file. By default, mod_python checks the time-stamp of the file and reloads the module if the module's file modification date is later than the last import or reload.

This options is useful in production environment where the modules do not change, it will save some processing time and give a small performance gain.

5.2.8 PythonOption

Syntax: PythonOption key value
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

Assigns a key value pair to a table that can be later retrieved by the `req.get_options()` function. This is useful to pass information between the apache configuration files ('httpd.conf', '.htaccess', etc) and the Python programs.

5.2.9 PythonPath

Syntax: PythonPath *path*
Context: server config, virtual host, directory, htaccess
Override: not None
Module: mod_python.c

PythonPath directive sets the PythonPath. The path must be specified in Python list notation, e.g.

```
PythonPath "['/usr/local/lib/python2.0', '/usr/local/lib/site_python', '/some/other/place']"
```

The path specified in this directive will replace the path, not add to it. However, because the value of the directive is evaled, to append a directory to the path, one can specify something like

```
PythonPath "sys.path+['/mydir']"
```

Mod_python tries to minimize the number of evals associated with the PythonPath directive because evals are slow and can negatively impact performance, especially when the directive is specified in an '.htaccess' file which gets parsed at every hit. Mod_python will remember the arguments to the PythonPath directive in the un-evaled form, and before evaling the value it will compare it to the remembered value. If the value is the same, no action is taken.

Because of this, you should not rely on the directive as a way to restore the pythonpath to some value if your code changes it.

Note that this directive should not be used as a security measure since the Python path is easily manipulated from within the scripts.

CHAPTER

SIX

Standard Handlers

6.1 Publisher Handler

The `publisher` handler is a good way to avoid writing your own handlers and focus on rapid application development. It was inspired by *Zope* ZPublisher.

6.1.1 Introduction

To use the handler, you need the following lines in your configuration

```
<Directory /some/path}
    SetHandler python-program
    PythonHandler mod_python.publisher
</Directory>
```

This handler allows access to functions and variables within a module via URL's. For example, if you have the following module, called 'hello.py':

```
""" Publisher example """

def say(req, what="NOTHING"):
    return "I am saying %s" % what
```

A URL `http://www.mysite.com/hello.py/say` would return '`I am saying NOTHING`'. A URL `http://www.mysite.com/hello.py/say?what=hello` would return '`I am saying hello`'.

6.1.2 The Publishing Algorithm

The Publisher handler maps a URI directly to a Python variable or callable object, then, respectively, returns it's string representation or calls it returning the string representation of the return value.

Traversal

The Publisher handler locates and imports the module specified in the URI. The module location is determined from the `Request.filename` attribute. Before importing, the file extension, if any, is discarded.

Once module is imported, the remaining part of the URI up to the beginning of any query data (a.k.a. PATH_INFO) is used to find an object within the module. The Publisher handler *traverses* the path, one element at a time from left to right, mapping the elements to Python object within the module.

The traversal will stop and `HTTP_NOTFOUND` will be returned to the client if:

- Any of the traversed object's names begin with an underscore ('_'). Use underscores to protect objects that should not be accessible from the web.
- A module is encountered. Published objects cannot be modules for security reasons.

If an object in the path could not be found, `HTTP_NOT_FOUND` is returned to the client.

Argument Matching and Invocation

Once the destination object is found, if it is callable and not a class, the Publisher handler will get a list of arguments that the object expects. This list is compared with names of fields from HTML form data submitted by the client via `POST` or `GET`. Values of fields whose names match the names of callable object arguments will be passed as strings.

If the destination is not callable or is a class, then its string representation is returned to the client.

Authentication

The publisher handler provides simple ways to control access to modules and functions.

At every traversal step, the Publisher handler checks for presence of `__auth__` and `__access__` attributes (in this order), as well as `__auth_realm__` attribute.

If `__auth__` is found and it is callable, it will be called with three arguments: the `Request` object, a string containing the user name and a string containing the password. If the return value of `__auth__` is false, then `HTTP_UNAUTHORIZED` is returned to the client (which will usually cause a password dialog box to appear).

If `__auth__` is a dictionary, then the user name will be matched against the key and the password against the value associated with this key. If the key and password do not match, `HTTP_UNAUTHORIZED` is returned. Note that this requires storing passwords as clear text in source code, which is not very secure.

`__auth__` can also be a constant. In this case, if it is false (i.e. `None`, `0`, `""`, etc.), then `HTTP_UNAUTHORIZED` is returned.

If there exists an `__auth_realm__` string, it will be sent to the client as Authorization Realm (this is the text that usually appears at the top of the password dialog box).

If `__access__` is found and it is callable, it will be called with two arguments: the `Request` object and a string containing the user name. If the return value of `__access__` is false, then `HTTP_FORBIDDEN` is returned to the client.

If `__access__` is a list, then the user name will be matched against the list elements. If the user name is not in the list, `HTTP_FORBIDDEN` is returned.

Similarly to `__auth__`, `__access__` can be a constant.

In the example below, only user "eggs" with password "spam" can access the `hello` function:

```
__auth_realm__ = "Members only"

def __auth__(req, user, passwd):

    if user == "eggs" and passwd == "spam" or \
       user == "joe" and passwd == "eoj":
        return 1
    else:
        return 0

def __access__(req, user):
    if user == "eggs":
        return 1
    else:
        return 0

def hello(req):
    return "hello"
```

Here is the same functionality, but using an alternative technique:

```
__auth_realm__ = "Members only"
__auth__ = {"eggs":"spam", "joe":"eoj"}
__access__ = ["eggs"]

def hello(req):
    return "hello"
```

Since functions cannot be assigned attributes, to protect a function, an `__auth__` or `__access__` function can be defined within the function, e.g.:

```
def sensitive(req):

    def __auth__(req, user, password):
        if user == 'spam' and password == 'eggs':
            # let them in
            return 1
        else:
            # no access
            return 0

    # something involving sensitive information
    return 'sensitive information'
```

Note that this technique will also work if `__auth__` or `__access__` is a constant, but will not work is they are a dictionary or a list.

The `__auth__` and `__access__` mechanisms exist independently of the standard *PythonAuthenHandler*. It is possible to use, for example, the handler to authenticate, then the `__access__` list to verify that the authenticated

user is allowed to a particular function.

NOTE: In order for mod_python to access `__auth__`, the module containing it must first be imported. Therefore, any module-level code will get executed during the import even if `__auth__` is false. To truly protect a module from being accessed, use other authentication mechanisms, e.g. the Apache `mod_auth` or with a mod_python *PythonAuthenHandler* handler.

6.1.3 Form Data

In the process of matching arguments, the Publisher handler creates an instance of *FieldStorage* class. A reference to this instance is stored in an attribute `form` of the `Request` object.

Since a `FieldStorage` can only be instantiated once per request, one must not attept to instantiate `FieldStorage` when using the Publisher handler and should use `Request.form` instead.

6.2 CGI Handler

CGI handler is a handler that emulates the CGI environment under mod_python.

Note that this is not a "true" CGI environment in that it is emulated at the Python level. `stdin` and `stdout` are provided by substituting `sys.stdin` and `sys.stdout`, and the environment is replaced by a dictionary. The implication is that any outside programs called from within this environment via `os.system`, etc. will not see the environment available to the Python program, nor will they be able to read/write from standard input/output with the results expected in a "true" CGI environment.

The handler is provided as a stepping stone for the migration of legacy code away from CGI. It is not recommended that you settle on using this handler as the preferred way to use mod_python for the long term.

To use it, simply add this to your '.htaccess' file:

```
SetHandler python-program
PythonHandler mod_python.cgihandler
```

As of version 2.7, the cgihandler will properly reload even indirectly imported modules. This is done by saving a list of loaded modules (sys.modules) prior to executing a CGI script, and then comparing it with a list of imported modules after the CGI script is done. Modules (except for whose whose `__file__` attribute points to the standard Python library location) will be deleted from sys.modules thereby forcing Python to load them again next time the CGI script imports them.

If you do not want the above behavior, edit the 'cgihandler.py' file and comment out the code delimited by ###.

Tests show the cgihandler leaking some memory when processing a lot of file uploads. It is still not clear what causes this. The way to work around this is to set the Apache `MaxRequestsPerChild` to a non-zero value.

6.3 Httpdapy handler

This handler is provided for people migrating from Httpdapy. To use it, add this to your `.htaccess` file:

```
PythonHandler mod_python.httpdapi
```

You will need to change one line in your code. Where it said

```
import httpdapi
```

it now needs to say

```
from mod_python import httpdapi
```

If you were using authentication, in your .htaccess, instead of:

```
AuthPythonModule modulename
```

use

```
PythonOption authhandler modulename
```

NB: Make sure that the old httpdapi.py and apache.py are not in your python path anymore.

6.4 ZHandler

NOTE: This handler is being phased out in favor of the *Publisher* handler described in Section 6.1.

This handler allows one to use the Z Object Publisher (formerly Bobo) with mod_python. This gives you the power of Zope Object Publishing along with the speed of mod_python. It doesn't get any better than this!

WHAT IS ZPublisher?

ZPublisher is a component of Zope. While I don't profess at Zope itself as it seems to be designed for different type of users than me, I do think that the ZPublisher provides an ingeniously simple way of writing WWW applications in Python.

A quick example do demonstrate the power of ZPublisher.

Suppose you had a file called zhello.py like this:

```
"""A simple Bobo application"""

def sayHello( name = "World" ):
    """ Sais Hello  (this comment is required)"""
    return "Hello %s!" % name
```

Notice it has one method defined in it. Through ZPublisher, that method can be invoked through the web via a URL similar to this:

http://www.domain.tld/site/zhello/sayHello and
http://www.domain.tld/site/zhello/sayHello?name=Joe

Note how the query keyword "name" converted to a keyword argument to the function.

If the above didn't "click" for you, go read the ZPublisher documentation at http://classic.zope.org:8080/Documentation/Reference/ObjectPublishingIntro for a more in-depth explanation.

QUICK START

1. Download and install Zope.
2. Don't start it. You're only interested in ZPublisher, and in order for it to work, Zope doesn't need to be running.
3. Pick a www directory where you want to use ZPublisher. For our purposes let's imagine it is accessible via http://www.domain.tld/site.
4. Make sure that the FollowSymLinks option is on for this directory in httpd.conf.
5. Make a symlink in this directory to the ZPublisher directory:

```
cd site
ln -s /usr/local/src/Zope-2.1.0-src/lib/python/ZPublisher .
```

6. Verify that it is correct:

```
ls -l
lrwxr-xr-x  1 uid group    53 Dec 13 12:15 ZPublisher -> /usr/local/src/Zope-2.1.0-src/lib/
```

7. Create an '.htaccess' file with this in it:

```
SetHandler python-program
PythonHandler mod_python.zhandler
PythonDebug
```

8. Create an above mentioned zhello.py file.
9. Look at http://www.domain.tld/site/zhello/sayHello?name=Joe

Noteworthy:

This module automatically reloads modules just like any other mod_python module. But modules that are imported by your code will not get reloaded. There are ways around having to restart the server for script changes to take effect. For example, let's say you have a module called mycustomlib.py and you have a module that imports it. If you make a changes to mycustomlib.py, you can force the changes to take effect by requesting http://www.domain.tld/site/mycustomlib/. You will get a server error, but mycustomelib should get reloaded.

P.S.: ZPublisher is not Zope, but only part of it. As of right now, as far as I know, Zope will not work with mod_python. This is because of locking issues. Older versions of Zope had no locking, so different children of apache would corrupt the database by trying to access it at the same time. Starting with version 2 Zope does have locking, however, it seems that the first child locks the database without ever releasing it and after that no other process can access it.

If this is incorrect, and you can manage to get Zope to work without problems, please send me an e-mail and I will correct this documentation.

APPENDIX

A

Windows Installation

Notes originally created by Enrique Vaamonde evaamo@loquesea.com

Your mileage may vary with these instructions

You need to have the following packages properly installed and configured in your system:

- Python 1.5.2 or 2.0
- Apache 1.3
- Winzip 6.x or later.

You need to download both the mod_python.dll and the mod_python-x.tgz (where x is the version number) files from the main page. Once you have all the things above mentioned we're good to go.

1. Installing mod_python libraries

   - Use Winzip to extract the distribution file (mod_python-x.tgz) into a temporary folder (i.e `C:\temp`):
   - NOTE: If Winzip shows this warning "Archive contains one file, should Winzip decompress it to a temporary folder?" just click on Yes, the content of the file should appear in Winzip right after.
   - Select all the files in Winzip and click on the Extract button, then type-in the path or just browse your way to the temporary folder and click extract.
   - Open your Windows Explorer and locate the temporary folder where you extracted the distribution file, you should have a new folder in your temporary folder (`C:\temp\mod_python-x`).
   - Move (or just drag & drop) the mod_python-x folder into the Python lib folder (i.e `C:\Program Files\Python\lib`).
   - Move the files in the folder lib inside the mod_python folder (`C:\Program Files\Python\lib\mod_python-x\lib\mod_python`) to the `C:\Program Files\Python\lib\mod_python` folder. It's safe to delete these folders we just emptied.

2. Integrating it with Apache

   Once the distribution file is correctly extracted and later moved into the Python directory, it's time to modify your Apache configuration (httpd.conf) and integrate the server with mod_python. These are a few steps we must do first:

   - Locate the file mod_python.dll that you downloaded before and move it to Apache's modules folder (i.e `C:\Program Files\Apache Group\Apache\modules`).

- Go to the Apache configuration folder (i.e `C:\Program Files\Apache Group\Apache\conf\`) and edit the httpd.conf file.

  Add the following line in the section ”Dynamic Shared Object (DSO) Support” of the httpd.conf file:

  ```
  LoadModule python_module modules/mod_python.dll
  ```

- Add the following lines in the section ScriptAlias and CGI of the httpd.conf:

  ```
  <Directory "<Your Document Root>/python">
     AddHandler python-program .py
     PythonHandler test
     PythonDebug on
  </Directory>
  ```

  NOTE: Replace the ¡Your Document Root¿ above with the Document Root you specified on the DocumentRoot directive in the Apache’s httpd.conf file.

- Last, create a folder under your Document Root called python.

3. Testing

   - Create a text file in the folder we created above and call it mptest.py (you can use Notepad for this).
   - Insert the following lines and save the file (Make sure it gets saved with the .py extension):

     ```
     from mod_python import apache

     def handler(req):
        req.content_type = "text/plain"
        req.send_http_header()
        req.write("Hello World!")
        return apache.OK
     ```

   - Make sure Apache is running (or launch it!) and then point your browser to the URL referring to the mptest.py, you should see ”Hello World!”.

That’s it, you’re ready to roll!! If you don’t see the ”Hello World!” message, the next section is for you.

APPENDIX

B

VMS installation

```
How to build and install mod_python on a VMS system

James Gessling <jgessling@yahoo.com> Fri, 3 Nov 2000

This assumes apache and python already installed successfully.  I tested
Compaq's CSWS version and 1.3.12 version's of Apache.  Python was 1.5.2 from
http://decus.decus.de/~zessin/python.

0) download current release (wrote this for 2.6.3) from www.modpython.org.

1) create directories on a VMS system something like:

dka0:[mod_python.src.include]

2) put the .c files in src, the .h in include

3) Cut the script off the end of this file, save it in the src directory.
Edit as necessary and use it to compile and link mod_python.exe.  Sorry,
I didn't make much effort to make it very sophisticated.

4) Under your python lib directory, add a subdirectory [.mod_python].

For example: dka100:[python.python-1_5_2.lib]

5) Populate this subdirectory with mod_python .py files.
This allows for module importing like:

    import mod_python.apache

which will find apache.py
```

```
6) Edit apache$root:[conf]httpd.conf to add line:

   Include /apache$root/conf/mod_python.conf

(typically at the end of the file)

7) create apache$root:[conf]mod_python.conf containing:

###########################################################################
##
# Mod_Python config
###########################################################################
##
#
# Load the dynamic MOD_PYTHON module
# note pythonpath must be in python list literal format
#
LoadModule PYTHON_MODULE modules/mod_python.exe

<Directory />
    AddHandler python-program .py
    PythonHandler mptest
    PythonDebug On
    PythonPath
"['/dka100/python/python-1_5_2/lib','/dka100/python/python-1_5_2/
vms/tools','/apache$root/htdocs/python']"
</Directory>
#

8) put mod_python.exe into apache$common:[modules] so it can be found and
loaded.  (create the directory if required).

9) fire up the web server with @sys$startup:apache$startup

10) Create a file mptest.py in a python subdirectory of your document root,
Typically apache$common:[htdocs.python].  Like this:

      from mod_python import apache

      def handler(req):
          req.send_http_header()
          req.write("Hello World!")
          return apache.OK

( watch your indenting, as usual )

11) point browser to:  http://node.place.com/python/mptest.py

12) enjoy "hello world"
```

```
$! build script, edit as needed to match the directories where your
$!               files are located.  Note /nowarning on cc, this is
$!               required because of a #define clash between apache
$!               and python.  If not used, the .exe is marked as
$!               having compilation warnings and won't load. Apache
$!               should already have been started to create apache$httpd_shr
$!               logical name,  Running the apache server with the -X flag
$!               as an interactive process can be used for debugging if
$!               necessary.
$ set noon
$ library/create mod_python_lib
$ cc :== cc  /nowarning/prefix=all/include=(dka100:[python.python-1_5_2],-
                        dka100:[python.python-1_5_2.include],-
                        dka0:[],-
                        dka200:[apache.apache.src.include],-
                        dka200:[apache.apache.src.os.openvms])
$ cc  _apachemodule
$ library/insert mod_python_lib  _apachemodule
$ cc  connobject
$ library/insert mod_python_lib  connobject
$ cc  mod_python
$ cc  requestobject
$ library/insert mod_python_lib  requestobject
$ cc  serverobject
$ library/insert mod_python_lib  serverobject
$ cc  tableobject
$ library/insert mod_python_lib  tableobject
$ cc  util
$ library/insert mod_python_lib  util
$!   mod_python
$ link/share/sysexe mod_python,sys$input/opt
SYMBOL_VECTOR=(PYTHON_MODULE=DATA)
mod_python_lib/lib
apache$httpd_shr/share
dka100:[python.python-1_5_2.vms.o_alpha]python_d00/lib
dka100:[python.python-1_5_2.vms.o_alpha]modules_d00/lib
dka100:[python.python-1_5_2.vms.o_alpha]vms_macro_d00/lib
dka100:[python.python-1_5_2.vms.o_alpha]objects_d00/lib
dka100:[python.python-1_5_2.vms.o_alpha]parser_d00/lib
dka100:[python.python-1_5_2.vms.o_alpha]vms_d00/lib
dka100:[python.python-1_5_2.vms.o_alpha]modules_d00/lib
dka100:[python.python-1_5_2.vms.o_alpha]vms_macro_d00/lib
dka100:[python.python-1_5_2.vms.o_alpha]vms_d00/lib
case_sensitive=no
$!
$ exit
```

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W