CherryPy on Apache2 with mod_python

Ferry Boender

Revision History

Revision 1.5 November 9, 2009 Revised by: FB

1. Introduction

I’ve recently written a web application using Python using the following libraries:

- CherryPy v3.0.2
- Mako v0.1.8
- SQLAlchemy v0.3.7

CherryPy has a built-in web server which you can use during development and for actually running the application. I already had an Apache with some old PHP programs however, so I couldn’t serve the Python web application using CherryPy’s built-in web server, cause I didn’t want to serve it on a port other than port 80. Fortunately, CherryPy applications can also be served with Apache using mod_python.

Setting up to run it through mod_python turned out to be somewhat of a major pain though. It took me a total of about 4 hours getting it to work. The information on the CherryPy website about mod_python turns out to be incorrect, incomplete and a little dated.

So in this article I’ll describe how I eventually managed to set up my application to work with both the built-in server as well as with Apache v2 and which pitfalls to look out for.

2. Install and enable mod_python

First off, you’ll have to install mod_python for Apache2. For Debian and Ubuntu systems, this is as easy as:

```
~# apt-get install libapache2-mod-python
```

If you’ve compiled Apache2 from source package, you’ll probably have to recompile or something. Please check the Apache2 manual or your Unix of choice for more information.

Next, you have to enable the mod_python module in Apache. This means you have to create a symlink to `/etc/apache2/mods-available/mod_python.load` in `/etc/apache2/mods-enabled` like this:

```
~# ln -s /etc/apache2 /etc/apache2/mods-available/mod_python.load /etc/apache2/mods-enabled
~# /etc/init.d/apache2 restart
```

Debian users can use the following command instead:

```
~# a2enmod mod_python
~# /etc/init.d/apache2 restart
```

3. Configure Apache

After enabling the mod_python module, we'll need to configure a directory to serve a Python program instead of 'normal\' contents. First off, I'll show you my personal setup so you know what you need to change.

The webroot I chose for the application is `/var/www/dvd.dev.local/htdocs`. The `/var/www/dvd.dev.local` directory contains two dirs: `logs` and `htdocs`. The application will live in the `htdocs` dir. The application will be served by its own virtual host: `dvd.dev.local`. The application itself is a simple DVD manage front-end, with a couple of tables in a SQLite database. The application is `dvd.py`.

As said, the application will live in the `/var/www/dvd.dev.local/htdocs`. Here's how that directory looks:

```
www-data@jib:~/dvd.dev.local/htdocs$ ls -Fl
total 20
-rw-r--r-- 1 www-data www-data 2048 Oct 10 11:10 dvd.db
-rw-r--r-- 1 www-data www-data  429 Oct 11 13:50 dvd.ini
-rwxr-xr-x 1 www-data www-data 3817 Oct 13 12:15 dvd.py*
drwxr-xr-x 2 www-data www-data 4096 Oct  6 12:08 images/
drwxr-xr-x 2 www-data www-data 4096 Oct  7 15:55 templates/
```

Now, we need to configure the virtual host and the htdocs directory so it will properly host the application. For this, we create a new virtual host configuration in `/etc/apache2/sites-available/dvd.dev.local`. This is the default way virtual host configurations are managed under Debian.

We place the following in the file:

```
<VirtualHost *:80>
        ServerAdmin webmaster@dvd.dev.local
        ServerName dvd.dev.local
```

```
DocumentRoot /var/www/dvd.dev.local/htdocs/

<Location />
        PythonPath "sys.path+['/var/www/dvd.dev.local/htdocs']"
        SetHandler python-program
        PythonHandler cherrypy._cpmodpy::handler
        PythonOption cherrypy.setup dvd::start_modpython
        PythonDebug On
</Location>

        LogLevel warn
        ErrorLog /var/www/dvd.dev.local/logs/error.log
        CustomLog /var/www/dvd.dev.local/logs/access.log combined
</VirtualHost>
```

I will only explain the `<Location />` part, because the rest is basic Apache stuff, and should be obvious. First we specify the `<Location />` directive. This refers to the DocumentRoot option, so it affects every request made for anything under `/var/www/dvd.dev.local/htdocs`. Now for the individual options in the Location directive:

- **PythonPath**: Instructs mod_python that is should append the directory `/var/www/dvd.dev.local/htdocs` to the sys.path of the Python interpreter. This is needed because mod_python will try to import your application as a module, so the directory with the application needs to be in the path.
- **SetHandler**: Instructs Apache that it should use mod_python to serve requests for URLs in this location. * **PythonHandler**: Instructs mod_python what it should run for each request. In this case, it should run the `handler()` function in the `_cpmodpy` file in the `cherrypy` module. This is a special method that's part of the CherryPy framework and allows you to run your CherryPy application with mod_python.
- **PythonOption**: PythonOption allows you to set arbitrary key/value pairs which can be read by the Python program being run. It's kind of like a configuration. Here, we set the `cherrypy.setup` option to the `start_modpython` function in the `dvd` module, which is our application. CherryPy's `_cpmodpy.handler()` will read this option and run the function defined in it for each request that is made.
- **PythonDebug**: This directive is set to On, so that mod_python will display errors, though it doesn't appear to work properly.

Enable the new virtual host by symlinking `/etc/apache2/sites-available/dvd.dev.local` to `/etc/apache2/sites-enabled/001-dvd.dev.local` (Debian/Ubuntu users can use the command: `a2ensite dvd.dev.local`).

4. How it works

When you start Apache, the following happens:

1. Apache is restarted.

2. Apache loads the mod_python module.
3. For each Apache child, mod_python starts a Python interpreter.

For each request that comes in, the following happens:

1. A request comes in for, say, `http://dvd.dev.local/`.
2. mod_python runs the cherrypy._cpmodpy.handler() method.
3. If this is the first request, the following happens. (This is only done once for each Apache child process, since the interpreter stays in memory).
   a. CherryPy sets up the CherryPy framework.
   b. Your application (dvd.py in this case) is imported.
   c. The function defined in the `cherrypy.setup` option PythonOption is read and run.
4. `cherrypy._cpmodpy.handler()` runs the request through the CherryPy framework like it normally does.
5. The output is sent to Apache, which returns it to the client.

5. Getting your application to work

Okay, so now it's time to get your application to work with mod_python. All that's needed is to put the CherryPy framework from blocking to non-blocking mode, and you're done.

Note: Philip Stark mentions that since CherryPy 3.1, you do not have to put CherryPy in non-blocking mode any longer. The "blocking=false" option has been deprecated and will generate an error in your log file. Keep on reading, however, cause there are still a lot of things to take care of.

At least, that's the theory. In practice, there's a whole lot more that needs to be done. There are a bunch of pitfalls that might prevent your application from running properly, and debugging them can be a real bitch.

5.1. Pitfalls

Here are a couple of things you should keep in mind when trying to get your application to work with mod_python:

- The application's working directory is `/`! This means that any referencing to files such as configurations or databases need to be absolute instead of relative. You always need to specify the entire path to the file.

- mod_python and CherryPy cache all kinds of stuff, so you'll have to reload your Apache for each change you make to source code. A `/etc/init.d/apache reload` will usually do the job. The easiest thing to do is first making sure your application works with the built-in web server by su-ing to the user as which your web server runs, changing the current working directory to `/`, and then run the application from there (`/var/www/dvd.dev.local/htdocs/dvd.py`).
- Error logging and displaying will fail unless everything is set up *exactly* right. Read on to learn how to minimize the number of problems you'll run into, and how to make errors appear as best as we can.
- Try getting your application to work in the root directory of the virtual host before trying to get it too work in a sub directory. Sub directories require additional configuration, and more configuration means more potential problems.

5.2. Error logging

The main thing to get right when trying to get your application to work under mod_python is error reporting. Without this, you'll be lost. Basically anything that causes an error will show the dreaded `Unrecoverable error in the server.` Here are some basic guidelines you can follow in order to make sure you receive errors or how you can evade them:

- Errors during the application start-up, such as syntax errors, are not shown in the logs, no matter what you try, so it's imperative that you get this right. If you get the Unrecoverable error, and you're not seeing any tracebacks anywhere, check the program by running it on the commandline!
- Make sure the error logs you specify everywhere are writable by the web user.
- Start CherryPy's error logging as soon as possible. (see below)
- Check all the logs for Python errors. Below I will show how to set up CherryPy's logging. You should check the logfile specified there. Also check the default Apache error log for the vhost. Other than that, check the log file for the default virtual host (that's the one you see when you make a request for the IP of your machine in your web browser). Also check the global Apache error log (/var/log/apache2/error.log under Debian). Errors may show up in *any* of these files!
- Make sure your application also runs okay when you're not running it from the directory the application is actually in. Run it as the web server user from the root directory like this: `su - www-data; cd /; /var/www/dvd.dev.local/htdocs/dvd.py` and see what happens. Fix any problems before attempting to run it through mod_python.

Now, it's important to start CherryPy's error handling as soon as possible in your application, especially if you're getting the `Unrecoverable` error. You can do this by doing the following as soon as possible in your application:

```
import cherrypy
cherrypy.config['log.error_file'] = '/var/www/dvd.dev.local/logs/py_error.log'
```

Make sure the log file exists, and is writable by the web server user. Don't just assume it is, just because the permissions are correct! Never assume, always verify:

```
[root@jib]/var/www/dvd.dev.local/logs# su - www-data
```

```
www-data@jib:~$ echo "test" >> /var/www/dvd.dev.local/logs/py_error.log
www-data@jib:~$ tail -n1 /var/www/dvd.dev.local/logs/py_error.log
test
```

5.3. Modifying your program to support mod_python

Okay, now you need to modify your application so it can be run both through mod_python as well as with CherryPy's built-in server. Below is a little boilerplate template based on my application.

```
#!/usr/bin/python

import os
import cherrypy
import mako.template, mako.lookup
import sqlalchemy.orm

#
# Specify a logfile as soon as possible so we increase the chance of logging
# errors. This could be commented out once the application works on the live
# server under mod_python
#
cherrypy.config['log.error_file'] = '/var/www/dvd.dev.local/logs/py_error.log'

#
# Our actual (dummy) web application
#
class DVD:
    def index(self, sort = None, edit_id = None):
        return('Hello world!')
    index.exposed = True

#
# Set up cherrypy so it's independent of the path being run from, and load the
# configuration.
#
path_base = os.path.dirname(__file__)
path_config = os.path.join(path_base, 'dvd.ini')
path_db = os.path.join(path_base, 'dvd.db')

#cherrypy.config.update(path_config)

#
# Set up stuff for our application to use.
#
metadata = sqlalchemy.BoundMetaData('sqlite://%s' % (path_db))

#
# These methods take care of running the application via either mod_python or
# stand-alone using the built-in CherryPy server.
#
```

```
def start_modpython():
    cherrypy.engine.SIGHUP = None
    cherrypy.engine.SIGTERM = None
    cherrypy.tree.mount(DVD(), config=path_config)
    cherrypy.engine.start(blocking=False)

def start_standalone():
    cherrypy.quickstart(DVD(), config=path_config)

#
# If we're not being imported, it means we should be running stand-alone.
#
if __name__ == '__main__':
    start_standalone()
```

The main things to note here are:

- We set up error logging as soon as possible, just below the imports. This is mostly used for debugging purposes; you can remove it once everything works properly.
- Since the current working directory for the script will be `/`, we need to refer to each and every file in an absolute fashion. We get the scripts directory using `os.path.dirname(__file__)`.
- Two methods are defined: `start_modpython()` and `start_standalone()`. These do as advertised: one starts the application in a way that is compatible with mod_python (non-blocking), the other starts the stand-alone version. `start_modpython()` is called directly by mod_python (`PythonOption cherrypy.setup dvd::start_modpython`). The other is started by looking at `__name__`. If the current file is being imported (mod_apache imports the application), the if loop at the end of the program will not be run. If the application is run from the commandline, the if loop will execute.

The boilerplate above can probably be improved on a bit. Many of the paths would normally be specified in the dvd.ini file.

The `dvd.ini` file looks like this:

```
[global]
log.screen = False
log.error_file = '/var/www/dvd.dev.local/logs/py_error.log'
show_tracebacks = True

tools.sessions.on = True
tools.sessions.storage_type = "file"
tools.sessions.storage_path = "/tmp/"
tools.sessions.timeout = 60

dvd.password = 'foo'
```

Explanation:

- **log.screen**: Log errors and access to stdout. For mod_python, this isn't needed but it is when running stand-alone, because otherwise you won't see errors. When you're developing the application, you can turn this on.
- **log.error_file**: This specifies where CherryPy should log errors and such. In the boilerplate above, we turn on logging immediately after doing our imports, so we can see all the errors in the log. You can remove this line from your Python program once the core of your application (the main body) works without errors. This configuration option will then make sure that errors that don't occur in the main body but in, say, your application's object (in this case `DVD()`) will still show up.

6. Serve in a sub directory

If you want to host the Python application in a sub directory under the document root (for instance, `http//dvd.dev.local/user1/`), you'll have to do some additional configurating. First of all, you have to modify the virtual host configuration:

```
<Location /user1>
    PythonPath "sys.path+['/var/www/dvd.dev.local/htdocs/user1']"
    SetHandler python-program
    PythonHandler cherrypy._cpmodpy::handler
    PythonOption cherrypy.setup dvd::start_modpython
    PythonDebug On
</Location>
```

In the example above, we've modified the normal virtual host configuration's `<Location>` and `PythonPath` directives to reflect the sub directory we want to run them in.

Next, we need to change our source code so that CherryPy knows it should expect the sub directory when running the application:

```
def start_modpython():
    cherrypy.engine.SIGHUP = None
    cherrypy.engine.SIGTERM = None
    cherrypy.tree.mount(DVD(), '/user1', config=path_config)
    cherrypy.engine.start(blocking=False)

def start_standalone():
    cherrypy.quickstart(DVD(), '/user1', config=path_config)
```

Here, we added a second parameter to the `tree.mount()` and `quickstart()` calls. Don't forget the leading `/`, or it won't work.

7. Serving static files

When serving static files (such as images) using the built-in web server, you specify something like the following in the configuration file (`dvd.ini`, in this case):

```
[/]
tools.staticdir.root = "/var/www/dvd.dev.local/htdocs/"

[/images]
tools.staticdir.on = True
tools.staticdir.dir = "images"
```

When running under mod_python, you can let CherryPy serve static files like it does when running stand-alone. But, you can also let Apache handle static files, which can be faster in some cases. Here's how to do that. Simply add the following to the virtual host configuration file:

```
<Location /images/>
    SetHandler None
</Location>
```

This will instruct Apache that anything found in the `/var/www/dvd.dev.local/htdocs/images/` dir shouldn't be handled by mod_python but by Apache itself. If you want to just unset the Python handling for files in this directory, you can use `SetHandler default-handler` instead, which will restore Apache's default handler. Do note though that the built-in CherryPy server doesn't support any of the Apache handlers, so doing this is probably not usefull, if not outright more dangerous.

8. About this document

8.1. Document License

Author:

```
Ferry Boender
<ferry (DOT) boender (AT) electricmonk (DOT) nl>
```

8.2. Acknowledgements

Many thanks to the following people for providing feedback and other help during the cherrypy-exploration and writing of this document:

- Michiel van Baak
- Fumanchu
- Philip Stark