mod_wsgi Documentation

Release 4.5.17

Graham Dumpleton

Jul 07, 2017

Contents

The mod_wsgi package implements a simple to use Apache module which can host any Python web application which supports the Python WSGI specification. The package can be installed in two different ways depending on your requirements.

The first is as a traditional Apache module installed into an existing Apache installation. Following this path you will need to manually configure Apache to load mod_wsgi and pass through web requests to your WSGI application.

The second way of installing mod_wsgi is to install it from PyPi using the Python `pip` command. This builds and installs mod_wsgi into your Python installation or virtual environment. The program `mod_wsgi-express` will then be available, allowing you to run up Apache with mod_wsgi from the command line with an automatically generated configuration. This approach does not require you to perform any configuration of Apache yourself.

Both installation types are suitable for production deployments. The latter approach using `mod_wsgi-express` is the best solution if wishing to use Apache and mod_wsgi within a Docker container to host your WSGI application. It is also a better choice when using mod_wsgi during the development of your Python web application as you will be able to run it directly from your terminal.

CHAPTER 1

Project Status

The mod_wsgi project is still being developed and maintained. The available time of the sole developer is however limited. As a result, progress may appear to be slow.

In general, the documentation is in a bit of a mess right now and somewhat outdated, so if you can't find something then ask on the mod_wsgi mailing list for help. Also check out the *Release Notes* as they at least are being updated.

A lot of the more recent changes are being made with the aim of making it a lot easier to deploy Apache with mod_wsgi in Docker based environments. Changes included the ability to install mod_wsgi using `pip`, along with an admin command called `mod_wsgi-express` which provides a really simple way of starting up Apache and mod_wsgi from the command line with an automatically generated configuration.

CHAPTER 2

Security Issues

Due to security issues in versions of mod_wsgi up to and including version 3.4, ensure that you are using version 3.5 or later.

Release notes for versions containing security related fixes are:

- *Version 3.5*

Because many Linux distributions still ship ancient out of date versions, which are not supported, it is highly recommended you avoid using packaged binary versions provided by your Linux distribution. Instead install mod_wsgi from source code, ensuring you keep up to date with the most recent version.

CHAPTER 3

Getting Started

If starting out with mod_wsgi it is recommended you start out with a simple 'Hello World' type application.

Do not attempt to use a Python web application dependent on a web framework such as Django, Flask or Pyramid until you have got a basic 'Hello World' application running first. The simpler WSGI application will validate that your mod_wsgi installation is working okay and that you at least understand the basics of configuring Apache.

You can find a simple 'Hello World' WSGI application, along with setup instructions for the traditional way of setting up Apache and mod_wsgi, described in the *Quick Configuration Guide*. For a bit more in-depth information and additional examples see the *Configuration Guidelines*.

Note that unless you are using Windows, where such a choice is not available, you should always use daemon mode of mod_wsgi. This is not the default mode, so you will need to ensure you follow the instructions to enable daemon mode.

For a simpler way of running a Python WSGI application using mod_wsgi, also checkout `mod_wsgi-express`, details of which can currently be found at:

> https://pypi.python.org/pypi/mod_wsgi

CHAPTER 4

Requirements

The mod_wsgi package can be compiled for and used with most recent patch revisions of Apache 2.0, 2.2 or 2.4 on UNIX like systems, such as Linux and MacOS X, as well as Windows.

It is highly recommended that you use Apache 2.4. Older versions of Apache have architectural design problems and sub optimal configuration defaults, that can result in excessive memory usage in certain circumstances. More recent mod_wsgi versions attempt to protect against these problems in Apache 2.0 and 2.2, however it is still better to use Apache 2.4.

Any of the single threaded 'prefork' or multithreaded 'worker' and 'event' Apache MPMs can be used when running on UNIX like systems.

Both Python 2 and 3 are supported. The minimum recommended versions of each being Python 2.6 and 3.3 respectively. The Python installation must have been installed in a way that shared libraries for Python are provided such that embedding of Python in another application is possible.

The mod_wsgi package should be able to host any Python web application which complies with the WSGI specification (PEP 3333). The implementation is very strict with its interpretation of the WSGI specification. Other WSGI servers available aren't as strict and allow Python web applications to run which do not comply with the WSGI specification. If your Python web application doesn't comply properly with the WSGI specification, then it may fail to run or may run sub optimally when using mod_wsgi.

CHAPTER 5

Installation

The mod_wsgi package can be installed from source code or may also be available as a pre built binary package as part of your Linux distribution.

Do be aware though that Linux distributions generally ship out of date versions of mod_wsgi and for long term support (LTS) versions of Linux can be anything up to about 5 years old. Those older versions are not supported in any way even though they are part of a so called LTS version of Linux.

If you want support and want to ensure you have the most up to date and bug free version of mod_wsgi, you should consider building and installing mod_wsgi from the source code.

For instructions on how to compile mod_wsgi from the source code for UNIX like operating systems such as Linux and MacOS X see:

- *Quick Installation Guide*
- *Installation On MacOS X*

If you are on Windows, you should instead use:

- https://github.com/GrahamDumpleton/mod_wsgi/blob/develop/win32/README.rst

CHAPTER 6

Troubleshooting

If you are having problems getting mod_wsgi to start up or do what you want it to do, first off ensure that you read the following documents:

- *Installation Issues*
- *Configuration Issues*
- *Application Issues*

You can also do some basic checking of your installation and configuration to validate that how it is setup is how you expect it to be. See the following document:

- *Checking Your Installation*

If none of the common issues match up with the problem you are seeing and are after other ideas, or you have the need to perform more low level debugging, check out the *User Guides*.

CHAPTER 7

User Guides

Quick Installation Guide

This document describes the steps for installing mod_wsgi on a UNIX system from the original source code.

Apache Requirements

Apache 2.0, 2.2 or 2.4 can be used.

For Apache 2.0, 2.2 and 2.4, the single threaded 'prefork' or multithreaded 'worker' Apache MPMs can be used. For Apache 2.4 the 'event' MPM can also be used.

The version of Apache and its runtime libraries must have be compiled with support for threading.

On Linux systems, if Apache has been installed from a package repository, you must have installed the corresponding Apache "dev" package as well.

For most Linux distributions, the "dev" package for Apache 2.X is "apache2-dev" where the corresponding Apache package was "apache2". Some systems however distinguish the "dev" package based on which MPM is used by Apache. As such, it may also be called "apache2-worker-dev" or "apache2-prefork-dev". If using Apache 2.X, do not mix things up and install "apache-dev" by mistake, which is the "dev" package for Apache 1.3 called just "apache".

Python Requirements

Any Python 2.X version from Python 2.6 onwards can be used. For Python 3.X, you will need Python 3.3 or later.

The version of Python being used must have been compiled with support for threading.

On Linux systems, if Python has been installed from a package repository, you must have installed the corresponding Python "dev" package as well.

Python should preferably be available as a shared library. If this is not the case then base runtime memory usage of mod_wsgi will be greater.

Unpacking The Source Code

Source code tar balls can be obtained from:

- https://github.com/GrahamDumpleton/mod_wsgi/releases

After having downloaded the tar ball for the version you want to use, unpack it with the command:

```
tar xvfz mod_wsgi-X.Y.tar.gz
```

Replace 'X.Y' with the actual version number for that being used.

Configuring The Source Code

To setup the package ready for building run the "configure" script from within the source code directory:

```
./configure
```

The configure script will attempt to identify the Apache installation to use by searching in various standard locations for the Apache build tools included with your distribution called "apxs2" or "apxs". If not found in any of these standard locations, your PATH will be searched.

Which Python installation to use will be determined by looking for the "python" executable in your PATH.

If these programs are not in a standard location, they cannot be found in your PATH, or you wish to use alternate versions to those found, the `--with-apxs` and `--with-python` options can be used in conjunction with the "configure" script:

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

On some Linux distributions, such as SUSE and CentOS, it will be necessary to use the `--with-apxs` option and specify either "/usr/sbin/apxs2-worker" or "/usr/sbin/apxs2-prefork". This is necessary as the Linux distribtions allow installation of "dev" packages for both Apache MPM variants at the same time, whereas other Linux distributions do not.

If you have multiple versions of Python installed and you are not using that which is the default, you may have to organise that the PATH inherited by the Apache application when run will result in Apache finding the alternate version. Alternatively, the WSGIPythonHome directive should be used to specify the exact location of the Python installation corresponding to the version of Python compiled against. If this is not done, the version of Python running within Apache may attempt to use the Python modules from the wrong version of Python.

Building The Source Code

Once the package has been configured, it can be built by running:

```
make
```

If the mod_wsgi source code does not build successfully, see:

- *Installation Issues*

If successful, the only product of the build process that needs to be installed is the Apache module itself. There are no separate Python code files as everything is done within C code compiled into the Apache module.

To install the Apache module into the standard location for Apache modules as dictated by Apache for your installation, run:

```
make install
```

Installation should be done as the 'root' user or 'sudo' command if appropriate.

If you want to install the Apache module in a non standard location dictated by how your operating system distribution structures the configuration files and modules for Apache, you will need to copy the file manually into place.

If installing the Apache module by hand, the file is called 'mod_wsgi.so'. The compiled Apache module can be found in the ".libs" subdirectory. The name of the file should be kept the same when copied into its appropriate location.

Loading Module Into Apache

Once the Apache module has been installed into your Apache installation's module directory, it is still necessary to configure Apache to actually load the module.

Exactly how this is done and in which of the main Apache configuration files it should be placed, is dependent on which version of Apache you are using and may also be influenced by how your operating system's Apache distribution has organised the Apache configuration files. You may therefore need to check with any documentation for your operating system to see in what way the procedure may need to be modified.

In the simplest case, all that is required is to add a line of the form:

```
LoadModule wsgi_module modules/mod_wsgi.so
```

into the main Apache "httpd.conf" configuration file at the same point that other Apache modules are being loaded. The last option to the directive should either be an absolute path to where the mod_wsgi module file is located, or a path expressed relative to the root of your Apache installation. If you used "make" to install the package, see where it copied the file to work out what to set this value to.

Restart Apache Web Server

Having adding the required directives you should perform a restart of Apache to check everything is okay. If you are using an unmodified Apache distribution from the Apache Software Foundation, a restart is performed using the 'apachectl' command:

```
apachectl restart
```

If you see any sort of problem, or if you are upgrading from an older version of mod_wsgi, it is recommended you actually stop and the start Apache instead:

```
apachectl stop
apachectl start
```

Note that on many Linux distributions where Apache is prepackaged, the Apache software has been modified and as a result the 'apachectl' command may not work properly or the command may not be present. On these systems, you will need to use whatever is the sanctioned method for restarting system services.

This may be via an 'init.d' script:

```
/etc/init.d/httpd stop
/etc/init.d/httpd start
```

or via some special service maintenance script.

On Debian derived distributions, restarting Apache is usually done via the 'invoke-rc.d' command:

```
invoke-rc.d apache2 stop
invoke-rc.d apache2 start
```

On RedHat derived distributions, restarting Apache is usually done via the 'service' command:

```
service httpd stop
service httpd start
```

In nearly all cases the scripts used to restart Apache will need to be run as the 'root' user or via 'sudo'.

In general, for any system where you are using a prepackaged version of Apache, it is wise to always check the documentation for that package or system to determine the correct way to restart the Apache service. This is because they often use a wrapper around 'apachectl', or replace it, with a script which performs additional actions.

If all is okay, you should see a line of the form:

```
Apache/2.4.8 (Unix) mod_wsgi/4.4.21 Python/2.7 configured
```

in the Apache error log file.

Cleaning Up After Build

To cleanup after installation, run:

```
make clean
```

If you need to build the module for a different version of Apache, you should run:

```
make distclean
```

and then rerun "configure" against the alternate version of Apache before attempting to run "make" again.

Installation On MacOS X

If you are using MacOS X, mod_wsgi can be compiled from source code against the standard versions of Python and Apache httpd server supplied with the operating system. To do this though you will first need to have installed the Xcode command line tools.

The Xcode command line tools package provides a C compiler, along with header files and support tools for the Apache httpd server. If you have already set up your system so as to be able to install additional Python packages which include C extensions, you likely will already have the Xcode command line tools.

Install Xcode command line tools

To install the Xcode command line tools you should run the command:

```
xcode-select --install
```

If this gives you back the error message:

```
xcode-select: error: command line tools are already installed, use "Software Update"␣
↪to install updates
```

then the tools have already been installed. As noted by the warning message, do make sure you have run a system software update to ensure that you have the latest versions of these tools.

If you do not already have the Xcode command line tools installed, running that `xcode-select` command should result in you being prompted to install them. This may ask you to provide the details of an administrator account along with the password for that account.

Note that it is not necessary to install the whole of the Xcode developer application from the MacOS X App Store, only the command line tools using `xcode-select`. If you have installed the Xcode developer application, still ensure that the command line tools are installed and ensure you have run the system software update.

Configuring and building mod_wsgi

If you are using the Python and Apache httpd server packages provided with the operating system, all you need to do to configure the mod_wsgi source code before building it is to run in the mod_wsgi source code directory:

```
./configure
```

This should yield output similar to:

```
checking for apxs2... no
checking for apxs... /usr/sbin/apxs
checking for gcc... gcc
checking whether the C compiler works... yes
checking for C compiler default output file name... a.out
checking for suffix of executables...
checking whether we are cross compiling... no
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes
checking for gcc option to accept ISO C89... none needed
checking for prctl... no
checking Apache version... 2.4.18
checking for python... /usr/bin/python
configure: creating ./config.status
config.status: creating Makefile
```

The `configure` script should show that it has detected `apxs` as being located at `/usr/sbin/apxs` and `python` as being at `/usr/bin/python`.

If you get different values for `apxs` and `python` then it means that you likely have a separate installation of Python or the Apache httpd server installed on your system. If this is the case, to ensure that you use the versions of Python and Apache httpd server provided with the operating system instead use the command:

```
./configure --with-python=/usr/bin/python --with-apxs=/usr/sbin/apxs
```

Once you have configured the source code by running `configure`, you can build mod_wsgi using the command:

```
make
```

This will compile the mod_wsgi source code and produce a single `mod_wsgi.so` file which then needs to be installed into a common location so that the Apache httpd server can use it.

Installing the mod_wsgi module

What you need to do to install the mod_wsgi module depends on which version of MacOS X you are using.

For the Apache httpd server provided by the operating system, the directory `/usr/libexec/apache2` is used to store the compiled modules. Prior to MacOS X El Capitan (10.11) this directory was writable and the mod_wsgi module could be installed here along with all the default modules. With the introduction of the System Integrity Protection (SIP) feature in MacOS X El Capitan this directory is not writable, not even to the root user.

Because of this, if you are using a version of MacOS X prior to MacOS X El Capitan (10.11) you can use the command:

```
sudo make install
```

to install the mod_wsgi module. As `sudo` is being run, you will be prompted for your password. The module will be installed into the directory `/usr/libexec/apache2`. Within the Apache httpd server configuration file you can then use the standard `LoadModule` line of:

```
LoadModule wsgi_module libexec/apache2/mod_wsgi.so
```

If however you are using MacOS X El Capitan (10.11) or later, the mod_wsgi module will need to be installed into a different location. If you don't and try to run just `sudo make install`, it will fail with the output:

```
./apxs -i -S LIBEXECDIR=/usr/libexec/apache2 -n 'mod_wsgi' src/server/mod_wsgi.la
/usr/share/httpd/build/instdso.sh SH_LIBTOOL='./libtool' src/server/mod_wsgi.la /usr/
↪libexec/apache2
./libtool --mode=install install src/server/mod_wsgi.la /usr/libexec/apache2/
libtool: install: install src/server/.libs/mod_wsgi.so /usr/libexec/apache2/mod_wsgi.
↪so
install: /usr/libexec/apache2/mod_wsgi.so: Operation not permitted
apxs:Error: Command failed with rc=4653056
.
make: *** [install] Error 1
```

The directory you use to install the mod_wsgi module is up to you, but one suggested option is that you use the directory `/usr/local/httpd/modules`. Just ensure that this isn't already used by a separate installation of the Apache httpd server.

To install the mod_wsgi module into this directory use the command:

```
sudo make install LIBEXECDIR=/usr/local/httpd/modules
```

The output from the command will be similar to:

```
mkdir -p /usr/local/httpd/modules
./apxs -i -S LIBEXECDIR=/usr/local/httpd/modules -n 'mod_wsgi' src/server/mod_wsgi.la
/usr/share/httpd/build/instdso.sh SH_LIBTOOL='./libtool' src/server/mod_wsgi.la /usr/
↪local/httpd/modules
./libtool --mode=install install src/server/mod_wsgi.la /usr/local/httpd/modules/
libtool: install: install src/server/.libs/mod_wsgi.so /usr/local/httpd/modules/mod_
↪wsgi.so
libtool: install: install src/server/.libs/mod_wsgi.lai /usr/local/httpd/modules/mod_
↪wsgi.la
libtool: install: install src/server/.libs/mod_wsgi.a /usr/local/httpd/modules/mod_
↪wsgi.a
libtool: install: chmod 644 /usr/local/httpd/modules/mod_wsgi.a
libtool: install: ranlib /usr/local/httpd/modules/mod_wsgi.a
libtool: install: warning: remember to run `libtool --finish /usr/libexec/apache2'
chmod 755 /usr/local/httpd/modules/mod_wsgi.so
```

The warning about needing to run `libtool --finish` can be ignored as it is not required for everything to work.

With the mod_wsgi module installed in this location, the `LoadModule` line in the Apache httpd configuration file should be:

```
LoadModule wsgi_module /usr/local/httpd/modules/mod_wsgi.so
```

Normal steps to then configure the Apache httpd server and mod_wsgi for your specific WSGI application would then be followed.

Quick Configuration Guide

This document describes the steps for configuring mod_wsgi for a basic WSGI application.

If you are setting up mod_wsgi for the very first time, it is highly recommended that you follow the examples in this document. Make sure that you at least get the examples running to verify that mod_wsgi is working correctly before attempting to install any WSGI applications of your own.

WSGI Application Script File

WSGI is a specification of a generic API for mapping between an underlying web server and a Python web application. WSGI itself is described by Python PEP 3333:

- http://www.python.org/dev/peps/pep-3333/

The purpose of the WSGI specification is to provide a common mechanism for hosting a Python web application on a range of different web servers supporting the Python programming language.

A very simple WSGI application, and the one which should be used for the examples in this document, is as follows:

```
def application(environ, start_response):
    status = '200 OK'
    output = b'Hello World!'

    response_headers = [('Content-type', 'text/plain'),
                        ('Content-Length', str(len(output)))]
    start_response(status, response_headers)

    return [output]
```

This sample application will need to be placed into what will be referred to as the WSGI application script file. For the examples presented here, the WSGI application will be run as the user that Apache runs as. As such, the user that Apache runs as must have read access to both the WSGI application script file and all the parent directories that contain it.

Note that mod_wsgi requires that the WSGI application entry point be called 'application'. If you want to call it something else then you would need to configure mod_wsgi explicitly to use the other name. Thus, don't go arbitrarily changing the name of the function. If you do, even if you set up everything else correctly the application will not be found.

Mounting The WSGI Application

There are a number of ways that a WSGI application hosted by mod_wsgi can be mounted against a specific URL. These methods are similar to how one would configure traditional CGI applications.

The main approach entails explicitly declaring in the main Apache configuration file the URL mount point and a reference to the WSGI application script file. In this case the mapping is fixed, with changes only being able to be made by modifying the main Apache configuration and restarting Apache.

When using mod_cgi to host CGI applications, this would be done using the ScriptAlias directive. For mod_wsgi, the directive is instead called WSGIScriptAlias:

```
WSGIScriptAlias /myapp /usr/local/www/wsgi-scripts/myapp.wsgi
```

This directive can only appear in the main Apache configuration files. The directive can be used at server scope but would normally be placed within the VirtualHost container for a particular site. It cannot be used within either of the Location, Directory or Files container directives, nor can it be used within a ".htaccess" file.

The first argument to the WSGIScriptAlias directive should be the URL mount point for the WSGI application. For this case the URL should not contain a trailing slash. The only exception to this is if the WSGI application is to be mounted at the root of the web server, in which case '/' would be used.

The second argument to the WSGIScriptAlias directive should be an absolute pathname to the WSGI application script file. It is into this file that the sample WSGI application code should be placed.

Note that an absolute pathname must be used for the WSGI application script file supplied as the second argument. It is not possible to specify an application by Python module name alone. A full path is used for a number of reasons, the main one being so that all the Apache access controls can still be applied to indicate who can actually access the WSGI application.

Because the Apache access controls will apply, if the WSGI application is located outside of any directories already configured to be accessible to Apache, it will be necessary to tell Apache that files within that directory can be used. To do this the Directory directive must be used:

```
<Directory /usr/local/www/wsgi-scripts>
Order allow,deny
Allow from all
</Directory>
```

Note that it is highly recommended that the WSGI application script file in this case NOT be placed within the existing DocumentRoot for your main Apache installation, or the particular site you are setting it up for. This is because if that directory is otherwise being used as a source of static files, the source code for your application might be able to be downloaded.

You also should not use the home directory of a user account, as to do that would mean allowing Apache to serve up any files in that account. In this case any misconfiguration of Apache could end up exposing your whole account for downloading.

It is thus recommended that a special directory be setup distinct from other directories and that the only thing in that directory be the WSGI application script file, and if necessary any support files it requires.

A complete virtual host configuration for this type of setup would therefore be something like:

```
<VirtualHost *:80>

    ServerName www.example.com
    ServerAlias example.com
    ServerAdmin webmaster@example.com

    DocumentRoot /usr/local/www/documents

    <Directory /usr/local/www/documents>
    Order allow,deny
    Allow from all
    </Directory>

    WSGIScriptAlias /myapp /usr/local/www/wsgi-scripts/myapp.wsgi

    <Directory /usr/local/www/wsgi-scripts>
```

```
    Order allow,deny
    Allow from all
    </Directory>

</VirtualHost>
```

After appropriate changes have been made Apache will need to be restarted. For this example, the URL 'http://www.example.com/myapp' would then be used to access the the WSGI application.

Note that you obviously should substitute the paths and hostname with values appropriate for your system.

Mounting At Root Of Site

If instead you want to mount a WSGI application at the root of a site, simply list '/' as the mount point when configuring the WSGIScriptAlias directive:

```
WSGIScriptAlias / /usr/local/www/wsgi-scripts/myapp.wsgi
```

Do note however that doing so will mean that any static files contained in the DocumentRoot will be hidden and requests against URLs pertaining to the static files will instead be processed by the WSGI application.

In this situation it becomes necessary to remap using the Alias directive, any URLs for static files to the directory containing them:

```
Alias /robots.txt /usr/local/www/documents/robots.txt
Alias /favicon.ico /usr/local/www/documents/favicon.ico

Alias /media/ /usr/local/www/documents/media/
```

A complete virtual host configuration for this type of setup would therefore be something like:

```
<VirtualHost *:80>

    ServerName www.example.com
    ServerAlias example.com
    ServerAdmin webmaster@example.com

    DocumentRoot /usr/local/www/documents

    Alias /robots.txt /usr/local/www/documents/robots.txt
    Alias /favicon.ico /usr/local/www/documents/favicon.ico

    Alias /media/ /usr/local/www/documents/media/

    <Directory /usr/local/www/documents>
    Order allow,deny
    Allow from all
    </Directory>

    WSGIScriptAlias / /usr/local/www/wsgi-scripts/myapp.wsgi

    <Directory /usr/local/www/wsgi-scripts>
    Order allow,deny
    Allow from all
    </Directory>

</VirtualHost>
```

After appropriate changes have been made Apache will need to be restarted. For this example, the URL 'http://www.example.com/' would then be used to access the the WSGI application.

Note that you obviously should substitute the paths and hostname with values appropriate for your system.

Delegation To Daemon Process

By default any WSGI application will run in what is called embedded mode. That is, the application will be hosted within the Apache worker processes used to handle normal static file requests.

When embedded mode is used, whenever you make changes to your WSGI application code you would generally have to restart the whole Apache web server in order for changes to be picked up. This can be inconvenient, especially if the web server is a shared resource hosting other web applications at the same time, or you don't have root access to be able to restart the server and rely on someone else to restart it.

On UNIX systems when running Apache 2.X, an option which exists with mod_wsgi and that avoids the need to restart the whole Apache web server when code changes are made, is to use what is called daemon mode.

In daemon mode a set of processes is created for hosting a WSGI application, with any requests for that WSGI application automatically being routed to those processes for handling.

When code changes are made and it is desired that the daemon processes for the WSGI application be restarted, all that is required is to mark the WSGI application script file as modified by using the 'touch' command.

To make use of daemon mode for WSGI applications hosted within a specific site, the WSGIDaemonProcess and WSGIProcessGroup directives would need to be defined. For example, to setup a daemon process group containing two multithreaded process one could use:

```
WSGIDaemonProcess example.com processes=2 threads=15
WSGIProcessGroup example.com
```

A complete virtual host configuration for this type of setup would therefore be something like:

```
<VirtualHost *:80>

    ServerName www.example.com
    ServerAlias example.com
    ServerAdmin webmaster@example.com

    DocumentRoot /usr/local/www/documents

    Alias /robots.txt /usr/local/www/documents/robots.txt
    Alias /favicon.ico /usr/local/www/documents/favicon.ico

    Alias /media/ /usr/local/www/documents/media/

    <Directory /usr/local/www/documents>
    Order allow,deny
    Allow from all
    </Directory>

    WSGIDaemonProcess example.com processes=2 threads=15 display-name=%{GROUP}
    WSGIProcessGroup example.com

    WSGIScriptAlias / /usr/local/www/wsgi-scripts/myapp.wsgi

    <Directory /usr/local/www/wsgi-scripts>
    Order allow,deny
    Allow from all
```

```
    </Directory>

</VirtualHost>
```

After appropriate changes have been made Apache will need to be restarted. For this example, the URL 'http://www.example.com/' would then be used to access the the WSGI application.

Note that you obviously should substitute the paths and hostname with values appropriate for your system.

As mentioned previously, the daemon processes will be shutdown and restarted automatically if the WSGI application script file is modified.

For the sample application presented in this document the whole application is in that file. For more complicated applications the WSGI application script file will be merely an entry point to an application being imported from other Python modules or packages. In this later case, although no change may be required to the WSGI application script file itself, it can still be touched to trigger restarting of the daemon processes in the event that any code in the separate modules or packages is changed.

Note that only requests for the WSGI application are handled within the context of the daemon processes. Any requests for static files are still handled within the Apache worker processes.

Debugging Any Problems

To debug any problems one should take note of the type of error response being returned, but more importantly one should look at the Apache error logs for more detailed descriptions of a specific problem.

Being new to mod_wsgi it is highly recommended that the default Apache LogLevel be increased from 'warn' to 'info':

```
LogLevel info
```

When this is done mod_wsgi will output additional information regarding when daemon processes are created, when Python sub interpreters related to a group of WSGI applications are created and when WSGI application script files are loaded and/or reloaded. This information can be quite valuable in determining what problem may be occuring.

Note that where the LogLevel directive may have been defined both in and outside of a VirualHost directive, due to the VirtualHost declaring its own error logs, both instances of the LogLevel directive should be changed.

This is because although the virtual host may have its own error log, some information is still logged to the main Apache error log and the LogLevel directive outside of the virtual host context needs to be changed for that additional information to be recorded.

In other words, even if the VirtualHost has its own error log file, also look in the main Apache error log file for information as well.

Configuration Guidelines

The purpose of this document is to detail the basic configuration steps required for running WSGI applications with mod_wsgi.

The WSGIScriptAlias Directive

Configuring Apache to run WSGI applications using mod_wsgi is similar to how Apache is configured to run CGI applications. To streamline this task however, an additional configuration directive called WSGIScriptAlias is provided.

Like the ScriptAlias directive for CGI scripts, the mod_wsgi directive combines together a number of steps so as to reduce the amount of configuration required.

The first way of using the WSGIScriptAlias directive to indicate the WSGI application to be used, is to associate a WSGI application against a specific URL prefix:

```
WSGIScriptAlias /myapp /usr/local/wsgi/scripts/myapp.wsgi
```

The last option to the directive in this case must be a full pathname to the actual code file containing the WSGI application. A trailing slash should never be added to the last option when it is referring to an actual file.

The WSGI application contained within the code file specified should be called 'application'. For example:

```
def application(environ, start_response):
    status = '200 OK'
    output = b'Hello World!'

    response_headers = [('Content-type', 'text/plain'),
                        ('Content-Length', str(len(output)))]
    start_response(status, response_headers)

    return [output]
```

Note that an absolute pathname to a WSGI script file must be provided. It is not possible to specify an application by Python module name alone. A full path is used for a number of reasons, the main one being so that all the Apache access controls can still be applied to indicate who can actually access the WSGI application. Because these access controls will apply, if the WSGI application is located outside of any directories already known to Apache, it will be necessary to tell Apache that files within that directory can be used. To do this the Directory directive must be used:

```
<Directory /usr/local/wsgi/scripts>
Order allow,deny
Allow from all
</Directory>
```

Note that Apache access control directives such as Order and Allow should nearly always be applied to Directory and never to a Location. Adding them to a Location would not be regarded as best practice and would potentially weaken the security of your Apache server, especially where the Location was for '/'.

As for CGI scripts and the ScriptAlias directive, it is not necessary to have used the Options directive to enable the ExecCGI directive. This is because it is automatically implied from the use of the WSGIScriptAlias directive that the script must be executable.

For WSGIScriptAlias, to mount a WSGI application at the root of the web site, simply use '/' as the mount point:

```
WSGIScriptAlias / /usr/local/wsgi/scripts/myapp.wsgi
```

If you need to mount multiple WSGI applications, the directives can be listed more than once. When this occurs, those occuring first are given precedence. As such, those which are mounted at what would be a sub URL to another WSGI application, should always be listed earlier:

```
WSGIScriptAlias /wiki /usr/local/wsgi/scripts/mywiki.wsgi
WSGIScriptAlias /blog /usr/local/wsgi/scripts/myblog.wsgi
WSGIScriptAlias / /usr/local/wsgi/scripts/myapp.wsgi
```

The second way of using the WSGIScriptAlias directive is to use it to map to a directory containing any number of WSGI applications:

```
WSGIScriptAlias /wsgi/ /usr/local/wsgi/scripts/
```

When this is used, the next part of the URL after the URL prefix is used to identify which WSGI application script file within the target directory should be used. Both the mount point and the directory path must have a trailing slash.

If you want WSGI application scripts to use an extension, but don't wish to have that extension appear in the URL, then it is possible to use the WSGIScriptAliasMatch directive instead:

```
WSGIScriptAliasMatch ^/wsgi/([^/]+) /usr/local/wsgi/scripts/$1.wsgi
```

In this case, any path information appearing after the URL prefix, will be mapped to a corresponding WSGI script file in the directory, but with a '.wsgi' extension. The extension would though not need to be included in the URL.

In all ways that the WSGIScriptAlias can be used, the target script is not required to have any specific extension type and in particular it is not necessary to use a '.py' extension just because it contains Python code. Because the target script is not treated exactly like a traditional Python module, if an extension is used, it is recommended that '.wsgi' be used rather than '.py'.

The Apache Alias Directive

Although the WSGIScriptAlias directive is provided, the traditional Alias directive can still be used to enable execution of WSGI applications for specific URLs. The equivalent such configuration for:

```
WSGIScriptAlias /wsgi/ /usr/local/wsgi/scripts/

<Directory /usr/local/wsgi/scripts>
Order allow,deny
Allow from all
</Directory>
```

using the Alias directive would be:

```
Alias /wsgi/ /usr/local/wsgi/scripts/

<Directory /usr/local/wsgi/scripts>
Options ExecCGI

SetHandler wsgi-script

Order allow,deny
Allow from all
</Directory>
```

The additional steps required in this case are to enable the ability to execute CGI like scripts using the Options directive and define the Apache handler as 'wsgi-script'.

If wishing to hold a mixture of static files, normal CGI scripts and WSGI applications within the one directory, the AddHandler directive can be used instead of the SetHandler directive to distinguish between the various resource types based on resource extension:

```
Alias /wsgi/ /usr/local/wsgi/scripts/

<Directory /usr/local/wsgi/scripts>
Options ExecCGI

AddHandler cgi-script .cgi
AddHandler wsgi-script .wsgi

Order allow,deny
```

```
Allow from all
</Directory>
```

For whatever extension you use to identify a WSGI script file, ensure that you do not have a conflicting definition for that extension marking it as a CGI script file. For example, if you previously had all '.py' files being handled as 'cgi-script', consider disabling that before marking '.py' file as then being handled as 'wsgi-script' file in same context. If both are defined in same context, which is used will depend on the order of the directives and the wrong handler may be selected.

Because an extension is required to determine whether a script should be processed as a CGI script versus a WSGI application, the extension would need to appear in the URL. If this is not desired, then add the MultiViews option and MultiviewsMatch directive:

```
Alias /wsgi/ /usr/local/wsgi/scripts/

<Directory /usr/local/wsgi/scripts>
Options ExecCGI MultiViews
MultiviewsMatch Handlers

AddHandler cgi-script .cgi
AddHandler wsgi-script .wsgi

Order allow,deny
Allow from all
</Directory>
```

Adding of MultiViews in this instance and allowing multiviews to match Apache handlers will allow the extension to be dropped from the URL. Provided that for each resource there is only one alternative, Apache will then automatically select either the CGI script or WSGI application as appropriate for that resource. Use of multiviews in this way would make it possible to transparently migrate from CGI scripts to WSGI applications without the need to change any URLs.

A benefit of using the AddHandler directive as described above, is that it also allows a directory index page or directory browsing to be enabled for the directory. To enable directory browsing add the Indexes option:

```
Alias /wsgi/ /usr/local/wsgi/scripts/

<Directory /usr/local/wsgi/scripts>
Options ExecCGI Indexes

AddHandler cgi-script .cgi
AddHandler wsgi-script .wsgi

Order allow,deny
Allow from all
</Directory>
```

If a directory index page is enabled, it may refer to either a static file, CGI or WSGI application. The DirectoryIndex directive should be used to designate what should be used for the index page:

```
Alias /wsgi/ /usr/local/wsgi/scripts/

<Directory /usr/local/wsgi/scripts>
Options ExecCGI Indexes

DirectoryIndex index.html index.wsgi index.cgi

AddHandler cgi-script .cgi
AddHandler wsgi-script .wsgi
```

```
Order allow,deny
Allow from all
</Directory>
```

Using AddHandler or SetHandler to configure a WSGI application can also be done from within the '.htaccess' file located within the directory which a URL maps to. This will however only be possible where the directory has been enabled to allow these directives to be used. This would be done using the AllowOverride directive and enabling FileInfo for that directory. It would also be necessary to allow the execution of scripts using the Options directive by listing ExecCGI:

```
Alias /site/ /usr/local/wsgi/site/

<Directory /usr/local/wsgi/site>
AllowOverride FileInfo
Options ExecCGI MultiViews Indexes
MultiviewsMatch Handlers

Order allow,deny
Allow from all
</Directory>
```

This done, the '.htaccess' file could then contain:

```
DirectoryIndex index.html index.wsgi index.cgi

AddHandler cgi-script .cgi
AddHandler wsgi-script .wsgi
```

Note that the DirectoryIndex directive can only be used to designate a simple WSGI application which returns a single page for when the URL maps to the actual directory. Because the DirectoryIndex directive is not applied when the URL has additional path information beyond the leading portion of the URL which mapped to the directory, it cannot be used as a means of making a complex WSGI application responding to numerous URLs appear at the root of a server.

When using the AddHandler directive, with WSGI applications identified by the extension of the script file, the only way to make the WSGI application appear as the root of the server is to perform on the fly rewriting of the URL internal to Apache using mod_rewrite. The required rules for mod_rewrite to ensure that a WSGI application, implemented by the script file 'site.wsgi' in the root directory of the virtual host, appears as being mounted on the root of the virtual host would be:

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^(.*)$ /site.wsgi/$1 [QSA,PT,L]
```

Do note however that when the WSGI application is executed for a request the 'SCRIPT_NAME' variable indicating what the mount point of the application was will be '/site.wsgi'. This will mean that when a WSGI application constructs an absolute URL based on 'SCRIPT_NAME', it will include 'site.wsgi' in the URL rather than it being hidden. As this would probably be undesirable, many web frameworks provide an option to override what the value for the mount point is. If such a configuration option isn't available, it is just as easy to adjust the value of 'SCRIPT_NAME' in the 'site.wsgi' script file itself:

```
def _application(environ, start_response):
    # The original application.
    ...

import posixpath
```

```
def application(environ, start_response):
    # Wrapper to set SCRIPT_NAME to actual mount point.
    environ['SCRIPT_NAME'] = posixpath.dirname(environ['SCRIPT_NAME'])
    if environ['SCRIPT_NAME'] == '/':
        environ['SCRIPT_NAME'] = ''
    return _application(environ, start_response)
```

This wrapper will ensure that 'site.wsgi' never appears in the URL as long as it wasn't included in the first place and that access was always via the root of the web site instead.

Application Configuration

If it is necessary or desired to be able to pass configuration information through to a WSGI application from the Apache configuration file, then the SetEnv directive can be used:

```
WSGIScriptAlias / /usr/local/wsgi/scripts/demo.wsgi

SetEnv demo.templates /usr/local/wsgi/templates
SetEnv demo.mailhost mailhost
SetEnv demo.debugging 0
```

Any such variables added using the SetEnv directive will be automatically added to the WSGI environment passed to the application when executed.

Note that the WSGI environment is passed upon each request to the application in the 'environ' argument of the application object. This environment is totally unrelated to the process environment which is kept in 'os.environ'. The SetEnv directive has no effect on 'os.environ' and there is no way through Apache configuration directives to affect what is in the process environment.

If needing to dynamically set variables based on some aspects of the request itself, the RewriteRule directive may also be useful in some cases as an avenue to set application configuration variables.

For example, to enable additional debug only when the client is connecting from the localhost, the following might be used:

```
SetEnv demo.debugging 0

RewriteEngine On
RewriteCond %{REMOTE_ADDR} ^127.0.0.1$
RewriteRule . - [E=demo.debugging:1]
```

More elaborate schemes involving RewriteMap could also be employed.

Where SetEnv and RewriteRule are insufficient, then any further application configuration should be injected into an application using a WSGI application wrapper within the WSGI application script file:

```
def _application(environ, start_response):
    ...

def application(environ, start_response):
    if environ['REMOTE_ADDR'] in ['127.0.0.1']:
        environ['demo.debugging'] = '1'
    return _application(environ, start_response)
```

User Authentication

As is the case when using CGI scripts with Apache, authorisation headers are not passed through to WSGI applications. This is the case, as doing so could leak information about passwords through to a WSGI application which should not be able to see them when Apache is performing authorisation.

Unlike CGI scripts however, when using mod_wsgi, the WSGIPassAuthorization directive can be used to control whether HTTP authorisation headers are passed through to a WSGI application in the `HTTP_AUTHORIZATION` variable of the WSGI application environment when the equivalent HTTP request headers are present. This option would need to be set to `On` if the WSGI application was to handle authorisation rather than Apache doing it:

```
WSGIPassAuthorization On
```

If Apache is performing authorisation and not the WSGI application, a WSGI application can still find out what type of authorisation scheme was used by checking the variable `AUTH_TYPE` of the WSGI application environment. The login name of the authorised user can be determined by checking the variable `REMOTE_USER`.

Hosting Of Static Files

When the WSGIScriptAlias directive is used to mount an application at the root of the web server for a host, all requests for that host will be processed by the WSGI application. If is desired for performance reasons to still use Apache to host static files associated with the application, then the Alias directive can be used to designate the files and directories which should be served in this way:

```
Alias /robots.txt /usr/local/wsgi/static/robots.txt
Alias /favicon.ico /usr/local/wsgi/static/favicon.ico

AliasMatch /([^/]*\.css) /usr/local/wsgi/static/styles/$1

Alias /media/ /usr/local/wsgi/static/media/

<Directory /usr/local/wsgi/static>
Order deny,allow
Allow from all
</Directory>

WSGIScriptAlias / /usr/local/wsgi/scripts/myapp.wsgi

<Directory /usr/local/wsgi/scripts>
Order allow,deny
Allow from all
</Directory>
```

When listing the directives, list those for more specific URLs first. In practice this shouldn't actually be required as the Alias directive should take precedence over WSGIScriptAlias, but good practice all the same.

Do note though that if using Apache 1.3, the Alias directive will only take precedence over WSGIScriptAlias if the mod_wsgi module is loaded prior to the mod_alias module. To ensure this, the LoadModule/AddModule directives are used.

Note that there is never a need to use SetHandler to reset the Apache content handler back to 'None' for URLs mapped to static files. That this is a requirement for mod_python is a short coming in mod_python, do not do the same thing for mod_wsgi.

Limiting Request Content

By default Apache does not limit the amount of data that may be pushed to the server via a HTTP request such as a POST. That this is the case means that malicious users could attempt to overload a server by attempting to upload excessively large amounts of data.

If a WSGI application is not designed properly and doesn't limit this itself in some way, and attempts to load the whole request content into memory, it could cause an application to exhaust available memory.

If it is unknown if a WSGI application properly protects itself against such attempts to upload excessively large amounts of data, then the Apache LimitRequestBody directive can be used:

```
LimitRequestBody 1048576
```

The argument to the LimitRequestBody should be the maxumum number of bytes that should be allowed in the content of a request.

When this directive is used, mod_wsgi will perform the check prior to actually passing a request off to a WSGI application. When the limit is exceeded mod_wsgi will immediately return the HTTP 413 error response without even invoking the WSGI application to handle the request. Any request content will not be read as the client connection will then be closed.

Note that the HTTP 413 error response page will be that defined by Apache, or as specified by the Apache ErrorDocument directive for that error type.

Defining Application Groups

By default each WSGI application is placed into its own distinct application group. This means that each application will be given its own distinct Python sub interpreter to run code within. Although this means that applications will be isolated and cannot in general interfere with the Python code components of each other, each will load its own copy of all Python modules it requires into memory. If you have many applications and they use a lot of different Python modules this can result in large process sizes.

To avoid large process sizes, if you know that applications within a directory can safely coexist and run together within the same Python sub interpreter, you can specify that all applications within a certain context should be placed in the same application group. This is indicated by using the WSGIApplicationGroup directive:

```
<Directory /usr/local/wsgi/scripts>
WSGIApplicationGroup admin-scripts

Order allow,deny
Allow from all
</Directory>
```

The argument to the WSGIApplicationGroup directive can in general be any unique name of your choosing, although there are also a number of special values which you can use as well. For further information about these special values see the more detailed documentation on the *WSGIApplicationGroup* directive. Two of the special values worth highlighting are:

%{GLOBAL}

The application group name will be set to the empty string.

Any WSGI applications in the global application group will always be executed within the context of the first interpreter created by Python when it is initialised. Forcing a WSGI application to run within the first interpreter can be necessary when a third party C extension module for Python has used the simplified threading API for manipulation of the Python GIL and thus will not run correctly within any additional sub interpreters created by Python.

%{ENV:variable}

The application group name will be set to the value of the named environment variable. The environment variable is looked-up via the internal Apache notes and subprocess environment data structures and (if not found there) via getenv() from the Apache server process.

In an Apache configuration file, environment variables accessible using the `%{ENV}` variable reference can be setup by using directives such as SetEnv and RewriteRule.

For example, to group all WSGI scripts for a specific user when using mod_userdir within the same application group, the following could be used:

```
RewriteEngine On
RewriteCond %{REQUEST_URI} ^/~([^/]+)
RewriteRule . - [E=APPLICATION_GROUP:~%1]

<Directory /home/*/public_html/wsgi-scripts/>
Options ExecCGI
SetHandler wsgi-script
WSGIApplicationGroup %{ENV:APPLICATION_GROUP}
</Directory>
```

Defining Process Groups

By default all WSGI applications will run in what is called 'embedded' mode. That is, the applications are run within Python sub interpreters hosted within the Apache child processes. Although this results in the best performance possible, there are a few down sides.

First off, embedded mode is not recommended where you are not adept at tuning Apache. This is because the default MPM settings are never usually suitable for Python web applications, instead being biased towards static file serving and PHP applications. If you run embedded mode without tuning the MPM settings, you can experience problems with memory usage, due to default number of processes being too many, and can also experience load spikes, due to how Apache performs lazy creation of processes to meet demand.

Secondly, embedded mode would not be suitable for shared web hosting environments as all applications run as the same user and through various means could interfere with each other.

Running multiple Python applications within the same process, even if separated into distinct sub interpreters also presents other challenges and problems. These include problems with Python extension modules not being implemented correctly such that they work from a secondary sub interpreter, or when used from multiple sub interpreters at the same time.

Where multiple applications, potentially owned by different users, need to be run, 'daemon' mode of mod_wsgi should instead be used. Using daemon mode, each application can be delegated to its own dedicated daemon process running just the WSGI application, with the Apache child processes merely acting as proxies for delivering the requests to the application. Any static files associated with the application would still be served up by the Apache child processes to ensure best performance possible.

To denote that a daemon process should be created the WSGIDaemonProcess directive is used. The WSGIProcessGroup directive is then used to delegate specific WSGI applications to execute within that daemon process:

```
WSGIDaemonProcess www.site.com threads=15 maximum-requests=10000

Alias /favicon.ico /usr/local/wsgi/static/favicon.ico

AliasMatch /([^/]*\.css) /usr/local/wsgi/static/styles/$1

Alias /media/ /usr/local/wsgi/static/media/
```

```
<Directory /usr/local/wsgi/static>
Order deny,allow
Allow from all
</Directory>

WSGIScriptAlias / /usr/local/wsgi/scripts/myapp.wsgi
WSGIProcessGroup www.site.com

<Directory /usr/local/wsgi/scripts>

Order allow,deny
Allow from all
</Directory>
```

Where Apache has been started as the `root` user, the daemon processes can optionally be run as a user different to that which the Apache child processes would normally be run as. The number of daemon processes making up the process group and whether they are single or multithreaded can also be controlled.

A further option which should be considered is that which dictates the maximum number of requests that a daemon process should be allowed to accept before the daemon process is shutdown and restarted. This should be used where there are problems with increasing memory use due to problems with the application itself or a third party extension module.

As a general recommendation it would probably be a good idea to use the maximum requests option when running large installations of packages such as Trac and MoinMoin. Any large web site based on frameworks such as Django, TurboGears and Pylons or applications which use a database backend may also benefit.

If an application does not shutdown cleanly when the maximum number of requests has been reached, it will be killed off after the shutdown timeout has expired. If this occurs on a regular basis you should run with more than a single daemon process in the process group such that the other process can still accept requests while the first is being restarted.

If the maximum requests option is not specified, then the daemon process will never expire and will only be restarted if Apache is restarted or the user explicitly signals it to restart.

For further information about the options that can be supplied to the WSGIDaemonProcess directive see the more detailed documentation for *WSGIDaemonProcess*. A few of the options which can be supplied to the WSGIDaemonProcess directive worth highlighting are:

user=name | user=#uid

> Defines the UNIX user _name_ or numeric user _uid_ of the user that the daemon processes should be run as. If this option is not supplied the daemon processes will be run as the same user that Apache would run child processes and as defined by the User directive.
>
> Note that this option is ignored if Apache wasn't started as the root user, in which case no matter what the settings, the daemon processes will be run as the user that Apache was started as.

group=name | group=#gid

> Defines the UNIX group _name_ or numeric group _gid_ of the primary group that the daemon processes should be run as. If this option is not supplied the daemon processes will be run as the same group that Apache would run child processes and as defined by the Group directive.
>
> Note that this option is ignored if Apache wasn't started as the root user, in which case no matter what the settings, the daemon processes will be run as the group that Apache was started as.

processes=num

Defines the number of daemon processes that should be started in this process group. If not defined then only one process will be run in this process group.

Note that if this option is defined as 'processes=1', then the WSGI environment attribute called 'wsgi.multiprocess' will be set to be True whereas not providing the option at all will result in the attribute being set to be False. This distinction is to allow for where some form of mapping mechanism might be used to distribute requests across multiple process groups and thus in effect it is still a multi-process application. If you need to ensure that 'wsgi.multiprocess' is False so that interactive debuggers will work, simply do not specify the 'processes' option and allow the default single daemon process to be created in the process group.

threads=num

Defines the number of threads to be created to handle requests in each daemon process within the process group.

If this option is not defined then the default will be to create 15 threads in each daemon process within the process group.

maximum-requests=nnn

Defines a limit on the number of requests a daemon process should process before it is shutdown and restarted. Setting this to a non zero value has the benefit of limiting the amount of memory that a process can consume by (accidental) memory leakage.

If this option is not defined, or is defined to be 0, then the daemon process will be persistent and will continue to service requests until Apache itself is restarted or shutdown.

Note that the name of the daemon process group must be unique for the whole server. That is, it is not possible to use the same daemon process group name in different virtual hosts.

If the WSGIDaemonProcess directive is specified outside of all virtual host containers, any WSGI application can be delegated to be run within that daemon process group. If the WSGIDaemonProcess directive is specified within a virtual host container, only WSGI applications associated with virtual hosts with the same server name as that virtual host can be delegated to that set of daemon processes.

When WSGIDaemonProcess is associated with a virtual host, the error log associated with that virtual host will be used for all Apache error log output from mod_wsgi rather than it appear in the main Apache error log.

For example, if a server is hosting two virtual hosts and it is desired that the WSGI applications related to each virtual host run in distinct processes of their own and as a user which is the owner of that virtual host, the following could be used:

```
<VirtualHost *:80>
ServerName www.site1.com
CustomLog logs/www.site1.com-access_log common
ErrorLog logs/ww.site1.com-error_log

WSGIDaemonProcess www.site1.com user=joe group=joe processes=2 threads=25
WSGIProcessGroup www.site1.com

...
</VirtualHost>

<VirtualHost *:80>
ServerName www.site2.com
CustomLog logs/www.site2.com-access_log common
ErrorLog logs/www.site2.com-error_log

WSGIDaemonProcess www.site2.com user=bob group=bob processes=2 threads=25
WSGIProcessGroup www.site2.com
```

```
...
</VirtualHost>
```

When using the WSGIProcessGroup directive, the argument to the directive can be either one of two special expanding variables or the actual name of a group of daemon processes setup using the WSGIDaemonProcess directive. The meaning of the special variables are:

%{GLOBAL}

> The process group name will be set to the empty string. Any WSGI applications in the global process group will always be executed within the context of the standard Apache child processes. Such WSGI applications will incur the least runtime overhead, however, they will share the same process space with other Apache modules such as PHP, as well as the process being used to serve up static file content. Running WSGI applications within the standard Apache child processes will also mean the application will run as the user that Apache would normally run as.

%{ENV:variable}

> The process group name will be set to the value of the named environment variable. The environment variable is looked-up via the internal Apache notes and subprocess environment data structures and (if not found there) via getenv() from the Apache server process. The result must identify a named process group setup using the WSGIDaemonProcess directive.

In an Apache configuration file, environment variables accessible using the *%{ENV}* variable reference can be setup by using directives such as SetEnv and RewriteRule.

For example, to select which process group a specific WSGI application should execute within based on entries in a database file, the following could be used:

```
RewriteEngine On
RewriteMap wsgiprocmap dbm:/etc/httpd/wsgiprocmap.dbm
RewriteRule . - [E=PROCESS_GROUP:${wsgiprocmap:%{REQUEST_URI}}]

WSGIProcessGroup %{ENV:PROCESS_GROUP}
```

Note that the WSGIDaemonProcess directive and corresponding features are not available on Windows or when running Apache 1.3.

Installation Issues

Although mod_wsgi is not a large package in itself, it depends on both Apache and Python to get it compiled and installed. Because Apache and Python are complicated systems in their own right, various problems can come up during installation of mod_wsgi. These problems can arise for various reasons, including an incomplete or suboptimal Python installation or presence of multiple Python versions.

The purpose of this document is to capture all the known problems that can arise regarding installation, including workarounds if available.

If you are having a problem which doesn't seem to be covered by this document, also make sure you see *Configuration Issues* and *Application Issues*.

Missing Python Header Files

In order to compile mod_wsgi from source code you must have installed the full Python distribution, including header files. On a Linux distribution where binary Python packages are split into a runtime package and a developer package,

the developer package is often not installed by default. This means that you will be missing the header files required to compile mod_wsgi from source code. An example of the error messages you will see if the developer package is not installed are:

```
mod_wsgi.c:113:20: error: Python.h: No such file or directory
mod_wsgi.c:114:21: error: compile.h: No such file or directory
mod_wsgi.c:115:18: error: node.h: No such file or directory
mod_wsgi.c:116:20: error: osdefs.h: No such file or directory
mod_wsgi.c:119:2: error: #error Sorry, mod_wsgi requires at least Python 2.3.0.
mod_wsgi.c:123:2: error: #error Sorry, mod_wsgi requires that Python supporting␣
↪thread.
```

To remedy the problem, install the developer package for Python corresponding to the Python runtime package you have installed. What the name of the developer package is can vary from one Linux distribution to another. Normally it has the same name as the Python runtime package with `-dev` appended to the package name. You will need to lookup up list of available packages in your packaging system to determine actual name of package to install.

Lack Of Python Shared Library

In the optimal case, when mod_wsgi is compiled the resulting Apache module should be less than 250 Kbytes in size. If this is not the case and the module is over 1MB in size, it indicates that the version of Python being used was not originally configured so as to produce a Python shared library and is instead only producing a static library.

Although the existance of only a static library for Python doesn't normally cause compilation of mod_wsgi to fail, it does mean that when 'libtool' is used to generate the mod_wsgi Apache module, that it has to embed the actual static library objects into the Apache module instead of it being used as a shared library.

The consequences of this are that when the mod_wsgi Apache module is loaded by Apache, the operating system dynamic linker has to perform address relocations on the Python library component of the mod_wsgi Apache module. Because these relocations require memory to be modified, the full Python library then becomes private memory to the process and not shared.

On a Linux system this need to perform the address relocations at runtime will immediately cause each Apache child process to bloat out in size by between 1 and 2MB. On a Solaris system, depending on which compiler is being used and which options, the amount of additional memory used can be 5MB or more.

To determine whether the compiled mod_wsgi module is making use of a shared library for Python, many UNIX systems provide the 'ldd' program. The output from running this on the 'mod_wsgi.so' file would be something like:

```
$ ldd mod_wsgi.so
 linux-vdso.so.1 =>  (0x00007fffeb3fe000)
 libpython2.5.so.1.0 => /usr/local/lib/libpython2.5.so.1.0 (0x00002adebf94d000)
 libpthread.so.0 => /lib/libpthread.so.0 (0x00002adebfcba000)
 libdl.so.2 => /lib/libdl.so.2 (0x00002adebfed6000)
 libutil.so.1 => /lib/libutil.so.1 (0x00002adec00da000)
 libc.so.6 => /lib/libc.so.6 (0x00002adec02dd000)
 libm.so.6 => /lib/libm.so.6 (0x00002adec0635000)
 /lib64/ld-linux-x86-64.so.2 (0x0000555555554000)
```

Note how there is a dependency listed on the '.so' file for Python. If this is not present then mod_wsgi is using a static Python library.

Although mod_wsgi will still work when compiled against a version of Python which only provides a static library, you are highly encouraged to ensure that your Python installation has been configured and compiled with the `--enable-shared` option to enable the production and use of a shared library for Python.

If rebuilding Python to generate a shared library, do make sure that the Python shared library, or a symlink to it appears in the Python 'config' directory of your Python installation. If the shared library doesn't appear here next to

the static version of the library, 'libtool' will not be able to find it and will still use the static version of the library. It is understood that the Python build process may not actually do this, so you may have to do it by hand.

To check, go to the Python 'config' directory of your Python installation and do a directory listing:

```
$ ls -las

   4 drwxr-sr-x  2 root staff    4096 2007-11-29 23:26 .
  20 drwxr-sr-x 21 root staff   20480 2007-11-29 23:26 ..
   4 -rw-r--r--  1 root staff    2078 2007-11-29 23:26 config.c
   4 -rw-r--r--  1 root staff    1446 2007-11-29 23:26 config.c.in
   8 -rwxr-xr-x  1 root staff    7122 2007-11-29 23:26 install-sh
7664 -rw-r--r--  1 root staff 7833936 2007-11-29 23:26 libpython2.5.a
  40 -rw-r--r--  1 root staff   38327 2007-11-29 23:26 Makefile
   8 -rwxr-xr-x  1 root staff    7430 2007-11-29 23:26 makesetup
   8 -rw-r--r--  1 root staff    6456 2007-11-29 23:26 python.o
  20 -rw-r--r--  1 root staff   17862 2007-11-29 23:26 Setup
   4 -rw-r--r--  1 root staff     368 2007-11-29 23:26 Setup.config
   4 -rw-r--r--  1 root staff      41 2007-11-29 23:26 Setup.local
```

If you only see a '.a' file for Python library, then either Python wasn't installed with the shared library, or the shared library was placed elsewhere. What appears to normally happen is that the shared library is actually placed in the 'lib' directory two levels above the Python 'config' directory. In that case you need to create a symlink in the 'config' directory to where the shared library is actually installed:

```
$ ln -s ../../libpython2.5.so .
```

Apart from the additional memory consumption when using a static library, it is also preferable that a shared library be used where it is possible that you will upgrade your Python installation to a newer patch revision. This is because if you upgrade Python to a newer patch revision but do not recompile mod_wsgi, mod_wsgi will still incorporate the older static Python library and will not pick up any changes from the newer version of Python. This will result in undefined behaviour as the Python library code may not match up with the Python code modules or external modules in the Python installation. If a Python shared library is used, this will not be a problem.

Multiple Python Versions

Where there are multiple versions of Python installed on a system and it is necessary to ensure that a specific version is used, the `--with-python` option can be supplied to 'configure' when installing mod_wsgi:

```
./configure --with-python=/usr/local/bin/python2.5
```

This may be necessary where for example the default Python version supplied with the system is an older version of Python. More specifically, it would be required where it isn't possible to replace the older version of Python outright due to operating system management scripts being dependent on the older version of Python and not working with newer versions of Python.

Where multiple versions of Python are present and are installed under the same directory, this should generally be all that is required. If however the newer version of Python you wish to use is in a different location, for example under '/usr/local', it is possible that when Apache is started that it will not be able find the Python library files for the version of Python you wish to use.

This can occur because the Python library when initialised determines where the Python installation resides by looking through directories specified in the 'PATH' environment variable for the 'python' executable and using that as base location for calculating installation prefix. Specifically, the directory above the directory containing the 'python' executable is taken as being the installation prefix.

When the Python which should be used is installed in a non standard location, then that 'bin' directory is unlikely to be in the 'PATH' used by Apache when it is started. As such, rather than find '/usr/local/bin/python' it would instead find '/usr/bin/python' and so use '/usr' rather than the directory '/usr/local/' as the installation prefix.

When this occurs, if under '/usr' there was no Python installation of the same version number as Python which should be used, then normally:

```
'import site' failed; use -v for traceback
```

would appear in the Apache error log file when Python is first being initialised within Apache. Any attempt to make a request against a WSGI application would also result in errors as no modules at all except for inbuilt modules, would be able to be found when an attempt is made to import them.

Alternatively, if there was a Python installation of the same version, albeit not the desired installation, then there may be no obvious issues on startup, but at run time you may find modules cannot be found when being imported as they are installed into a different location than that which was being used. Even if equivalent module is found, it could fail at run time in subtle ways if the two Python installations are of same version but at the different locations are compiled in different ways, or if it is a third party module and they are different versions and so API is different.

In this situation it will be necessary to explicitly tell mod_wsgi where the Python executable for the version of Python which should be used, is located. This can be done using the WSGIPythonHome directive:

```
WSGIPythonHome /usr/local
```

The value given to the WSGIPythonHome directive should be a normalised path corresponding to that defined by the Python {{{sys.prefix}}} variable for the version of Python being used and passed to the `--with-python` option when configuring mod_wsgi:

```
>>> import sys
>>> sys.prefix
'/usr/local'
```

An alternative, although less desirable way of achieving this is to set the 'PATH' environment variable in the startup scripts for Apache. For a standard Apache installation using ASF structure, this can be done by editing the 'envvars' file in same directory as the Apache executable and adding the alternate bin directory to the head of the 'PATH':

```
PATH=/usr/local/bin:$PATH
export PATH
```

If there are any concerns over what Python installation directory is being used and you want to verify what it is, then use a small test WSGI script which outputs the values of 'sys.prefix' and 'sys.path'. For example:

```
import sys

def application(environ, start_response):
    status = '200 OK'
    output = b'Hello World!'

    response_headers = [('Content-type', 'text/plain'),
                        ('Content-Length', str(len(output)))]
    start_response(status, response_headers)

    print >> sys.stderr, 'sys.prefix = %s' % repr(sys.prefix)
    print >> sys.stderr, 'sys.path = %s' % repr(sys.path)

    return [output]
```

Using ModPython and ModWsgi

Using mod_python and mod_wsgi together is no longer supported and recent versions of mod_wsgi will cause the startup of Apache to be aborted if both are loaded at the same time.

Python Patch Level Mismatch

If the Python package is upgraded to a newer patch level revision, one will likely see the following warning messages in the Apache error log when Python is being initialised:

```
mod_wsgi: Compiled for Python/2.4.1.
mod_wsgi: Runtime using Python/2.4.2.
```

The warning is indicating that a newer version of Python is now being used than what mod_wsgi was originally compiled for.

This would generally not be a problem provided that both versions of Python were originally installed with the `--enable-shared` option supplied to 'configure'. If this option is used then the Python library will be linked in dynamically at runtime and so an upgrade to the Python version will be automatically used.

If `--enable-shared` was however not used and the Python library is therefore embedded into the actual mod_wsgi Apache module, then there is a risk of undefined behaviour. This is because the version of the Python library embedded into the mod_wsgi Apache module will be older than the corresponding Python code modules and extension modules being used from the Python library directory.

Thus, if a shared library is not being used for Python it will be necessary to rebuild mod_wsgi against the newer patch level revision of mod_wsgi and reinstall it.

Mixing 32 Bit And 64 Bit Packages

When attempting to compile mod_wsgi on a Linux system using an X86 64 bit processor, the following error message can arise:

```
/bin/sh /usr/lib64/apr/build/libtool --silent --mode=link gcc -o \
  mod_wsgi.la -I/usr/local/include/python2.4 -DNDEBUG  -rpath \
  /usr/lib64/httpd/modules -module -avoid-version mod_wsgi.lo \
  -L/usr/local/lib/python2.4/config -lpython2.4 -lpthread -ldl -lutil
/usr/bin/ld: /usr/local/lib/python2.4/config/
libpython2.4.a(abstract.o): relocation R_X86_64_32 against `a local
symbol' can not be used when making a shared object; recompile with -fPIC
/usr/local/lib/python2.4/config/libpython2.4.a: could not read symbols: Bad value
collect2: ld returned 1 exit status
apxs:Error: Command failed with rc=65536
.
make: *** [mod_wsgi.la] Error 1
```

This error is believed to be result of the version of Python being used having been originally compiled for the generic X86 32 bit architecture whereas mod_wsgi is being compiled for X86 64 bit architecture. The actual error arises in this case because 'libtool' would appear to be unable to generate a dynamically loadable module for the X86 64 bit architecture from a X86 32 bit static library. Alternatively, the problem is due to 'libtool' on this platform not being able to create a loadable module from a X86 64 bit static library in all cases.

If the first issue, the only solution to this problem is to recompile Python for the X86 64 bit architecture. When doing this, it is preferable, and may actually be necessary, to ensure that the `--enable-shared` option is provided to the 'configure' script for Python when it is being compiled and installed.

If rebuilding Python to generate a shared library, do make sure that the Python shared library, or a symlink to it appears in the Python 'config' directory of your Python installation. If the shared library doesn't appear here next to the static version of the library, 'libtool' will not be able to find it and will still use the static version of the library. It is understood that the Python build process may not actually do this, so you may have to do it by hand.

If the version of Python being used was compiled for X86 64 bit architecture and a shared library does exist, but not in the 'config' directory, then adding the missing symlink may be all that is required.

Unable To Find Python Shared Library

When mod_wsgi is built against a version of Python providing a shared library, the Python shared library must be in a directory which is searched for libraries at runtime by Apache. If this isn't the case the Python shared library will not be able to be found when loading the mod_wsgi module in to Apache. The error in this situation will be similar to:

```
error while loading shared libraries: libpython2.4.so.1.0: \
 cannot open shared object file: No such file or directory
```

A number of alternatives exist for resolving this problem. The preferred solution would be to copy the Python shared library into a directory which is searched for dynamic libraries at run time. Directories which would generally always be searched are '/lib' and '/usr/lib'.

For some systems the directory '/usr/local/lib' may also be searched, but this may depend on the directory having been explicitly added to the approrpiate system file listing the directories to be searched. The name and location of this configuration file differs between platforms. On Linux systems it is often called '/etc/ld.so.conf'. If changes are made to the file on Linux systems the 'ldconfig' command also needs to be run. See the manual page for 'ldconfig' for further details.

Rather than changing the system wide list of directories to search for shared libraries, additional search directories can be specified just for Apache. On Linux this would entail setting the 'LD_LIBRARY_PATH' environment variable to include the directory where the Python shared library is installed.

The setting and exporting of the environment variable would be placed in the Apache 'envvars' file, for a standard Apache installation, located in the same directory as the Apache web server executable. If using a customised Apache installation, such as on Red Hat, the 'envvars' file may not exist. In this case you would need to add this into the actual startup script for Apache. For Red Hat this is '/etc/sysconfig/httpd'.

A final alternative on some systems is to embed the directory to search for the Python shared library into the mod_wsgi Apache module itself. On Linux systems this can be done by setting the environment variable 'LD_RUN_PATH' to the directory containing the Python shared library when initially building the mod_wsgi source code.

GNU C Stack Smashing Extensions

Various Linux distributions are starting to ship with a version of the GNU C compiler which incorporates an extension which implements protection for stack-smashing. In some instances where such a compiler is used to build mod_wsgi, the module is unable to then be loaded by Apache. The specific problem is that the symbol `__stack_chk_fail_local` is being flagged as undefined:

```
$ invoke-rc.d apache2 reload
apache2: Syntax error on line 190 of /etc/apache2/apache2.conf: \
 Cannot load /usr/lib/apache2/modules/mod_wsgi.so into server: \
 /usr/lib/apache2/modules/mod_wsgi.so: \
 undefined symbol: __stack_chk_fail_local failed!
invoke-rc.d: initscript apache2, action "reload" failed.
```

The exact reason for this is not known but it is speculated to be caused when the system libraries or Apache itself has not been compiled with a version of the GNU C compiler incorporating the extension.

To workaround the problem, modify the 'Makefile' for mod_wsgi and change the value of 'CFLAGS' to:

```
CFLAGS = -Wc,-fno-stack-protector
```

Perform a 'clean' in the directory and then rebuild and reinstall the mod_wsgi module.

Undefined 'forkpty' On Fedora 7

On Fedora 7, the provided binary version of Apache is not linked against the 'libutil' system library. This causes problems when Python is initialised and the 'posix' module imported for the first time. This is because the 'posix' module requires functions from 'libutil' but they will not be present. The error encountered would be similar to:

```
httpd: Syntax error on line 54 of /etc/httpd/conf/httpd.conf: Cannot \
 load /etc/httpd/modules/mod_wsgi.so into server: \
 /etc/httpd/modules/mod_wsgi.so: undefined symbol: forkpty
```

This problem can be fixed by adding `-lutil` to the list of libraries to link mod_wsgi against when it is being built. This can be done by adding `-lutil` to the 'LDLIBS' variable in the mod_wsgi 'Makefile' after having run 'configure'.

An alternative method which may work is to edit the 'envvars' file, if it exists and is used, located in the same directory as the Apache 'httpd' executable, or the Apache startup script, and add:

```
LD_PRELOAD=/usr/lib/libutil.so
export LD_PRELOAD
```

Missing Include Files On SUSE

SUSE Linux follows a slightly different convention to other Linux distributions and has split their Apache "dev" packages in a way as to allow packages for different Apache MPMs to be installed at the same time. Although the resultant mod_wsgi module isn't strictly MPM specific, it does indirectly include the MPM specific header file "mpm.h". Because the header file is MPM specific, when configuring mod_wsgi, it is necessary to reference the version of "apxs" from the MPM specific "dev" package else the "mpm.h" header file will not be found at compile time. These errors are:

```
In file included from mod_wsgi.c:4882: /usr/include/apache2/mpm_common.h:46:17:␣
↪error: mpm.h: No such file or directory
...
mod_wsgi.c: In function 'wsgi_set_accept_mutex':
mod_wsgi.c:5200: error: 'ap_accept_lock_mech' undeclared (first use in this function)
mod_wsgi.c:5200: error: (Each undeclared identifier is reported only once
mod_wsgi.c:5200: error: for each function it appears in.)
apxs:Error: Command failed with rc=65536
```

To avoid this problem, when configuring mod_wsgi, it is necessary to use the `--with-apxs` option to designate that either "apxs2-worker" or "apxs2-prefork" should be used. Thus:

```
./configure --with-apxs=/usr/sbin/apxs2-worker
```

or:

```
./configure --with-apxs=/usr/sbin/apxs2-prefork
```

Although which is used is not important, since mod_wsgi when compiled isn't specific to either, best to use that which corresponds to the version of Apache being used.

Apache Maintainer Mode

When building mod_wsgi from source code, on UNIX systems there should be minimal if no compiler warnings. If you see a lot of warnings, especially complaints about `ap_strstr`, then your Apache installation has been configured for maintainer mode:

```
mod_wsgi.c: In function 'wsgi_process_group':
mod_wsgi.c:722: warning: passing argument 1 of 'ap_strstr' discards
qualifiers from pointer target type
mod_wsgi.c:740: warning: passing argument 1 of 'ap_strstr' discards
qualifiers from pointer target type
```

Specifically, whoever built the version of Apache being used supplied the option `--enable-maintainer-mode` when configuring Apache prior to installation. You would be able to tell at the time of compiling mod_wsgi if this has been done as the option `-DAP_DEBUG` would be supplied to the compiler when mod_wsgi source code is compiled.

These warnings can be ignored, but in general you shouldn't run Apache in maintainer mode.

A further reason for not running Apache in maintainer mode is that certain situations can cause Apache to fail an internal assertion check when using mod_wsgi. The specific error message is:

```
[crit] file http_filters.c, line 346, assertion "readbytes > 0" failed
[notice] child pid 18551 exit signal Aborted (6)
```

This occurs because the Apache code has an overly agressive assertion check, which is arguably incorrect. This particular assertion check will fail when a zero length read is perform on the Apache 'HTTP_IN' input filter.

This scenario can arise in mod_wsgi due to a workaround in place to get around a bug in Apache related to generation of '100-continue' response. The Apache bug is described in:

- https://issues.apache.org/bugzilla/show_bug.cgi?id=38014

The scenario can also be triggered as a result of a WSGI application performing a zero length read on 'wsgi.input'.

Changes to mod_wsgi are being investigated to see if zero length reads can be ignored, but due to the workaround for the bug, this would only be able to be done for Apache 2.2.8 or later.

The prefered solution is simply not to use Apache with maintainer mode enabled for systems where you are running real code. Unfortunately, it looks like some Linux distributions, eg. SUSE, accidentally released Apache binary packages with this mode enabled by default. You should update to a Apache binary package that doesn't have the mode enabled, or compile from source code.

Configuration Issues

Many Linux distributions in particular do not structure an Apache installation in the default manner as dictated by the original Apache code distributed by the Apache Software Foundation. This fact, and differences between different operating systems and distributions means that the configuration for mod_wsgi may sometimes have to be tweaked.

The purpose of this document is to capture all the known problems that can arise in respect of configuration.

If you are having a problem which doesn't seem to be covered by this document, also make sure you see *Installation Issues* and *Application Issues*.

Location Of UNIX Sockets

When mod_wsgi is used in 'daemon' mode, UNIX sockets are used to communicate between the Apache child processes and the daemon processes which are to handle a request.

These sockets and any related mutex lock files will be placed in the standard Apache runtime directory. This is the same directory that the Apache log files would normally be placed.

For some Linux distributions, restrictive permissions are placed on the standard Apache runtime directory such that the directory is not readable to others. This can cause problems with mod_wsgi because the user that the Apache child processes run as will subsequently not have the required permissions to access the directory to be able to connect to the sockets.

When this occurs, a '503 Service Temporarily Unavailable' error response would be received by the client. The Apache error log file would show messages of the form:

```
(13)Permission denied: mod_wsgi (pid=26962): Unable to connect to WSGI \
 daemon process '<process-name>' on '/etc/httpd/logs/wsgi.26957.0.1.sock' \
 after multiple attempts.
```

To resolve the problem, the WSGISocketPrefix directive should be defined to point at an alternate location. The value may be a location relative to the Apache root directory, or an absolute path.

On systems which restrict access to the standard Apache runtime directory, they normally provide an alternate directory for placing sockets and lock files used by Apache modules. This directory is usually called 'run' and to make use of this directory the WSGISocketPrefix directive would be set as follows:

```
WSGISocketPrefix run/wsgi
```

Although this may be present, do be aware that some Linux distributions, notably RedHat, also lock down the permissions of this directory as well so not readable to processes running as a non root user. In this situation you will be forced to use the operating system level '/var/run' directory rather than the HTTP specific directory:

```
WSGISocketPrefix /var/run/wsgi
```

Note, do not put the sockets in the system temporary working directory. That is, do not go making the prefix '/tmp/wsgi'. The directory should be one that is only writable by 'root' user, or if not starting Apache as 'root', the user that Apache is started as.

Application Issues

Although installation and configuration of mod_wsgi may be successful, there are a range of issues that can impact on specific WSGI applications. These problems can arise for various reasons, including conflicts between an application and other Apache modules or non WSGI applications hosted by Apache, a WSGI application not being portable, use of Python modules that are not fully compatible with the way that mod_wsgi uses Python sub interpreters, or dependence on a specific operating system execution environment.

The purpose of this document is to capture all the known problems that can arise, including workarounds if available, related to the actual running of a WSGI application.

Note that the majority of these issues are not unique to mod_wsgi and would also affect mod_python as well. This is because they arise due to the fact that the Python interpreter is being embedded within the Apache server itself. Unlike mod_python, in mod_wsgi there are ways of avoiding many of the problems by using daemon mode.

If you are having a problem which doesn't seem to be covered by this document, also make sure you see *Installation Issues* and *Configuration Issues*.

Access Rights Of Apache User

For most Apache installations the web server is initially started up as the root user. This is necessary as operating systems will block non root applications from making use of Internet ports below 1024. A web server responding to

HTTP and HTTPS requests on the standard ports will need to be able to acquire ports 80 and 443.

Once the web server has acquired these ports and forked off child processes to handle any requests, the user that the child processes run as will be switched to a non privileged user. The actual name of this user varies from one system to another with some commonly used names being 'apache', 'httpd', 'www', and 'wwwserv'. As well as the user being switched, the web server will also normally switch to an alternate group.

If running a WSGI application in embedded mode with mod_wsgi, the user and group that the Apache child processes run as will be inherited by the application. To determine which user and group would be used the main Apache configuration files should be consulted. The particular configuration directives which control this are `User` and `Group`. For example:

```
User www
Group www
```

Because this user is non privileged and will generally be different to the user that owns the files for a specific WSGI application, it is important that such files and the directories which contain them are accessible to others. If the files are not readable or the directories not searchable, the web server will not be able to see or read the files and execution of the WSGI application will fail at some point.

As well as being able to read files, if a WSGI application needs to be able to create or edit files, it will be necessary to create a special directory which it can use to create files in and which is owned by the same user that Apache is running as. Any files contained in the directory which it needs to edit should also be owned by the user that Apache is run as, or group privileges used in some way to ensure the application will have the required access to update the file.

One example of where access rights can be a problem in Python is with Python eggs which need to be unpacked at runtime by a WSGI application. This issue arises with Trac because of its ability for plugins to be packaged as Python eggs. Pylons with its focus on being able to support Python eggs in its deployment mechanism can also be affected. Because of the growing reliance on Python eggs however, the issue could arise for any WSGI application where you have installed Python eggs in their zipped up form rather than their unpacked form.

If your WSGI application is affected by this problem in relation to Python eggs, you would generally see a Python exception similar to the following occuring and being logged in the Apache error logs:

```
ExtractionError: Can't extract file(s) to egg cache

The following error occurred while trying to extract file(s) to the
Python egg cache:

[Errno 13] Permission denied: '/var/www/.python-eggs'

The Python egg cache directory is currently set to:

  /var/www/.python-eggs

Perhaps your account does not have write access to this directory?
You can change the cache directory by setting the PYTHON_EGG_CACHE
environment variable to point to an accessible directory.
```

To avoid this particular problem you can set the 'PYTHON_EGG_CACHE' cache environment variable at the start of the WSGI application script file. The environment variable should be set to a directory which is owned and/or writable by the user that Apache runs as:

```
import os
os.environ['PYTHON_EGG_CACHE'] = '/usr/local/pylons/python-eggs'
```

Alternatively, if using mod_wsgi 2.0, one could also use the WSGIPythonEggs directive for applications running in embedded mode, or the 'python-eggs' option to the WSGIDaemonProcess directive when using daemon mode.

Note that you should refrain from ever using directories or files which have been made writable to anyone as this could compromise security. Also be aware that if hosting multiple applications under the same web server, they will all run as the same user and so it will be possible for each to both see and modify each others files. If this is an issue, you should host the applications on different web servers running as different users or on different systems. Alternatively, any data required or updated by the application should be hosted in a database with separate accounts for each application.

Issues related to access rights can in general be avoided if daemon mode of mod_wsgi is used to run a WSGI application. This is because in daemon mode the user and group that the processes run as can be overridden and set to alternate values. Do however note additional issues related to 'HOME' environment variable as described below.

Secure Variants Of UNIX

In addition to the constraints imposed by Apache running as a distinct user, some variants of UNIX have features whereby access privileges for a specific user may be even further restricted. One example of such a system is SELinux. In such a system, the user that Apache runs as is typically restricted to only being able to access quite specific parts of the file system as well as possibly other resources or operating system library features.

If running such a system you will need to change the configuration for the security system to allow both mod_wsgi and you application to do what is required.

As an example, the extra security checks of such a system may present problems if the version of Python you are using only provides a static library and not a shared library. If you experience an error similar to:

```
Cannot load /etc/httpd/modules/mod_wsgi.so into server: \
 /etc/httpd/modules/mod_wsgi.so: cannot restore segment prot after reloc: \
 Permission denied
```

you will either need to configure the security system appropriately to allow that memory relocations in static code to work, or you would need to make sure that you reinstall Python such that it provides a shared library and rebuild mod_wsgi. Other issues around only having a static variant of the Python library available are described in section 'Lack Of Python Shared Library' of *Installation Issues*.

Even where a shared library is used, SELinux has also resulted in similar memory related errors when loading C extension modules at run time for Python:

```
ImportError: /opt/python2.6/lib/python2.6/lib-dynload/itertools.so: \
 failed to map segment from shared object: Permission denied
```

All up, configuring SELinux is a bit of a black art and so you are wise to do your research.

For some information about using mod_wsgi in a SELinux enabled environment check out:

- http://www.packtpub.com/article/selinux-secured-web-hosting-python-based-web-applications
- http://www.globalherald.net/jb01/weblog/21.html
- http://blog.endpoint.com/2010/02/selinux-httpd-modwsgi-26-rhel-centos-5.html

If you suspect that an issue may be caused by SELinux, you could temporarily try disabling it and doing a restart to verify whether it is the cause, but always re-enable it and do not disable it completely.

Application Working Directory

When Apache is started it is typically run such that the current working directory for the application is the root directory, although the actual directory may vary dependent on the system or any extra security system in place.

Importantly, the current working directory will generally never have any direct relationship to any specific WSGI application. As a result, an application should never assume that it can use relative path names for accessing the filesystem. All paths used should always be absolute path names.

An application should also never change the current working directory and then assume that it can then use relative paths. This is because other applications being hosted on the same web server may assume they can do the same thing with the result that you can never be sure what the current working directory may actually be.

You should not even assume that it is safe to change the working directory immediately prior to a specific operation, as use of multithreading can mean that another application could change it even before you get to perform the operation which depended on the current working directory being the value you set it to.

In the case of Python, if needing to use relative paths in order to make it easier to relocate an application, one can determine the directory that a specific code module is located in using `os.path.dirname(__file__)`. A full path name can then be constructed by using `os.path.join()` to merge the relative path with the directory name where the module was located.

Another option is to take the directory part of the `SCRIPT_FILENAME` variable from the WSGI environment as the base directory. The only other alternative is to rely on a centralised configuration file so that all absolute path names are at least defined in the one place.

Although it is preferable that an application never make assumptions about what the current working directory is, if for some reason the application cannot be changed the daemon mode of mod_wsgi could be used. This will work as an initial current working directory for the process can be specified as an option to the WSGIDaemonProcess directive used to configure the daemon process. Because the working directory applies to the whole process however, only the application requiring this working directory should be delegated to run within the context of that daemon process.

Application Environment Variables

When Python sub interpreters are created, each has its own copy of any modules which are loaded. They also each have their own copy of the set of environment variables inherited by the process and found in `os.environ`.

Problems can arise with the use of `os.environ` though, due to the fact that updates to `os.environ` are pushed back into the set of process environment variables. This means that if the Python sub interpreter which corresponds to another application group is created after `os.environ` has been updated, the new value for that environment variable will be inherited by the new Python sub interpreter.

This would not generally be a problem where a WSGI application is configured using a single mandatory environment variable, as the WSGI application script file for each application instance would be required to set it, thereby overriding any value inherited from another application instance via the process environment variables.

As example, Django relies on the `DJANGO_SETTINGS_MODULE` environment variable being set to be the name of the Python module containing Django's configuration settings. So long as each WSGI script file sets this variable all will be okay.

Where use of environment variables can be problematic though is where there are multiple environment variables that can be set, with some being optional and non overlapping sets of variables are used to configure different modes.

As example, Trac can be configured to host a single project by setting the `TRAC_ENV` environment variable. Alternatively, Trac can be configured to host a group of projects by setting the `TRAC_ENV_PARENT_DIR` environment variable. If both variables are set at the same time, then `TRAC_ENV` takes precedence.

If now within the one process you have a Trac instance of each type in different Python sub interpreters, if that using `TRAC_ENV` loads first, when the other is loaded it will inherit `TRAC_ENV` from the first and that will override `TRAC_ENV_PARENT_DIR`. The end result is that both sites point at the same single project, rather than the first being for the single project and the other being the group of projects.

Because of this potential leakage of environment variables between Python sub interpreters, it is preferable that WSGI applications not rely on environment variables for configuration.

A further reason that environment variables should not be used for configuration is that it then becomes impossible to host two instances of the same WSGI application component within the same Python sub interpreter if each would require a different value be set for the same environment variable. Note that this also applies to other means of hosting WSGI applications besides mod_wsgi and is not mod_wsgi specific.

As a consequence, because Django relies on the `DJANGO_SETTINGS_MODULE` environment variable being set to be the name of the Python module containing Django's configuration settings, it would be impossible to host two Django instances in the same Python sub interpreter. It is thus important that where there are multiple instances of Django that need to be run on the same web server, that they run in separate Python sub interpreters.

As it stands the default behaviour of mod_wsgi is to run different WSGI application scripts within the context of different Python sub interpreters. As such, this limitation in Django does not present as an immediate problem, however it should be kept in mind when attempting to merge multiple WSGI applications into one application group under one Python sub interpreter to try and limit memory use by avoiding duplicate instances of modules in memory.

The prefered way of configuring a WSGI application is for the application to be a class instance which at the point of initialisation is provided with its configuration data as an argument. Alternatively, or in conjunction with this, configuration information can be passed through to the WSGI application in the WSGI environment. Variables in the WSGI environment could be set by a WSGI middleware component, or from the Apache configuration files using the `SetEnv` directive.

Configuring an application when it is first constructed, or by supplying the configuration information through the WSGI environment variables, is thus the only way to ensure that a WSGI application is portable between different means of hosting WSGI applications. These problems can also be avoided by using daemon mode of mod_wsgi and delegating each WSGI application instance to a distinct daemon process group.

Timezone and Locale Settings

More insidious than the problem of leakage of application environment variable settings between sub interpreters, is where an environment variable is required by operating system libraries to set behaviour.

This is a problem because applications running in different sub interpreters could set the process environment variable to be different values. Rather than each seeing behaviour consistant with the setting they used, all applications will see behaviour reflecting the setting as determined by the last application to initialise itself.

Process environment variables where this can be a problem are the 'TZ' environment variable for setting the timezone, and the 'LANG', 'LC_TYPE', 'LC_COLLATE', 'LC_TIME' and 'LC_MESSAGES' environment variables for setting the locale and language settings.

The result of this, is that you cannot host multiple WSGI applications in the same process, even if running in different sub interpreters, if they require different settings for timezone, locale and/or language.

In this situation you would have no choice but to use mod_wsgi daemon mode and delegate applications requiring different settings to different daemon process groups. Alternatively, completely different instances of Apache should be used.

User HOME Environment Variable

If Apache is started automatically as 'root' when a machine is first booted it would inherit the user 'HOME' environment variable setting of the 'root' user. If however, Apache is started by a non privileged user via the 'sudo' command, it would inherit the 'HOME' environment variable of the user who started it, unless the `-H` option had been supplied to 'sudo'. In the case of the `-H` option being supplied, the 'HOME' environment variable of the 'root' user would again be used.

Because the value of the 'HOME' environment variable can vary based on how Apache has been started, an application should not therefore depend on the 'HOME' environment variable.

Unfortunately, parts of the Python standard library do use the 'HOME' environment variable as an authoritative source of information. In particular, the 'os.expanduser()' function gives precedence to the value of the 'HOME' environment variable over the home directory as obtained from the user password database entry:

```
if 'HOME' not in os.environ:
    import pwd
    userhome = pwd.getpwuid(os.getuid()).pw_dir
else:
    userhome = os.environ['HOME']
```

That the 'os.expanduser()' function does this means it can yield incorrect results. Since the 'setuptools' package uses 'os.expanduser()' on UNIX systems to calculate where to store Python EGGS, the location it tries to use can change based on who started Apache and how.

The only way to guarantee that the 'HOME' environment variable is set to a sensible value is for it to be set explicitly at the start of the WSGI script file before anything else is done:

```
import os, pwd
os.environ["HOME"] = pwd.getpwuid(os.getuid()).pw_dir
```

In mod_wsgi 2.0, if using daemon mode the value of the 'HOME' environment variable will be automatically reset to correspond to the home directory of the user that the daemon process is running as. This is not done for embedded mode however, due to the fact that the Apache child processes are shared with other Apache modules and it is not seen as appropriate that mod_wsgi should be changing the same environment that is used by these other unrelated modules.

For some consistency in the environment inherited by applications running in embedded mode, it is therefore recommended that 'sudo -H' at least always be used when restarting Apache from a non root account.

Application Global Variables

Because the Python sub interpreter which hosts a WSGI application is retained in memory between requests, any global data is effectively persistent and can be used to carry state forward from one request to the next. On UNIX systems however, Apache will normally use multiple processes to handle requests and each such process will have its own global data.

This means that although global data can be used, it can only be used to cache data which can be safely reused within the context of that single process. You cannot use global data as a means of holding information that must be visible to any request handler no matter which process it runs in.

If data must be visible to all request handlers across all Apache processes, then it will be necessary to store the data in the filesystem directly, or using a database. Alternatively, shared memory can be employed by using a package such as memcached.

Because your WSGI application can be spread across multiple process, one must also be very careful in respect of local caching mechanisms employed by database connector objects. If such an adapter is quite agressive in its caching, it is possible that a specific process may end up with an out of date view of data from a database where one of the other processes has since changed the data. The result may be that requests handled in different processes may give different results.

The problems described above can be alleviated to a degree by using daemon mode of mod_wsgi and restricting to one the number of daemon processes in the process group. This will ensure that all requests are serviced by the same process. If the data is only held in memory, it would however obviously be lost when Apache is restarted or the daemon process is restarted due to a maximum number of requests being reached.

Writing To Standard Output

No WSGI application component which claims to be portable should write to standard output. That is, an application should not use the Python `print` statement without directing output to some alternate stream. An application should also not write directly to `sys.stdout`.

This is necessary as an underlying WSGI adapter hosting the application may use standard output as the means of communicating a response back to a web server. This technique is for example used when WSGI is hosted within a CGI script.

Ideally any WSGI adapter which uses `sys.stdout` in this way should cache a reference to `sys.stdout` for its own use and then replace it with a reference to `sys.stderr`. There is however nothing in the WSGI specification that requires this or recommends it, so one can't therefore rely on it being done.

In order to highlight non portable WSGI application components which write to or use standard output in some way, mod_wsgi prior to version 3.0 replaced `sys.stdout` with an object which will raise an exception when any attempt is made to write to or make use of standard output:

```
IOError: sys.stdout access restricted by mod_wsgi
```

If the WSGI application you are using fails due to use of standard output being restricted and you cannot change the application or configure it to behave differently, you have one of two options. The first option is to replace `sys.stdout` with `sys.stderr` at the start of your WSGI application script file:

```
import sys
sys.stdout = sys.stderr
```

This will have the affect of directing any data written to standard output to standard error. Such data sent to standard error is then directed through the Apache logging system and will appear in the main Apache error log file.

The second option is to remove the restriction on using standard output imposed by mod_wsgi using a configuration directive:

```
WSGIRestrictStdout Off
```

This configuration directive must appear at global scope within the Apache configuration file outside of any VirtualHost container directives. It will remove the restriction on using standard output from all Python sub interpreters that mod_wsgi creates. There is no way using the configuration directive to remove the restriction from only one Python sub interpreter.

When the restriction is not imposed, any data written to standard output will also be directed through the Apache logging system and will appear in the main Apache error log file.

Ideally though, code should never use the 'print' statement without redirecting the output to 'sys.stderr'. Thus if the code can be changed, then it should be made to use something like:

```
import sys

def function():
    print >> sys.stderr, "application debug"
        ...
```

Also, note that code should ideally not be making assumptions about the environment it is executing in, eg., whether it is running in an interactive mode, by asking whether standard output is a tty. In other words, calling 'isatty()' will cause a similar error with mod_wsgi. If such code is a library module, the code should be providing a way to specifically flag that it is a non interactive application and not use magic to determine whether that is the case or not.

For further information about options for logging error messages and other debugging information from a WSGI application running under mod_wsgi see section 'Apache Error Log Files' of *Debugging Techniques*.

WSGI applications which are known to write data to standard output in their default configuration are CherryPy and TurboGears. Some plugins for Trac also have this problem. Thus one of these two techniques described above to remove the restriction, should be used in conjunction with these WSGI applications. Alternatively, those applications will need to be configured not to output log messages via standard output.

Note that the restrictions on writing to stdout were removed in mod_wsgi 3.0 because it was found that people couldn't be bothered to fix their code. Instead they just used the documented workarounds, thereby propogating their non portable WSGI application code. As such, since people just couldn't care, stopped promoting the idea of writing portable WSGI applications.

Reading From Standard Input

No general purpose WSGI application component which claims to be portable should read from standard input. That is, an application should not read directly from `sys.stdin` either directly or indirectly.

This is necessary as an underlying WSGI adapter hosting the application may use standard input as the means of receiving a request from a web server. This technique is for example used when WSGI is hosted within a CGI script.

Ideally any WSGI adapter which uses `sys.stdin` in this way should cache a reference to `sys.stdin` for its own use and then replace it with an instance of `StringIO.StringIO` wrapped around an empty string such that reading from standard input would always give the impression that there is no input data available. There is however nothing in the WSGI specification that requires this or recommends it, so one can't therefore rely on it being done.

In order to highlight non portable WSGI application components which try and read from or otherwise use standard input, mod_wsgi prior to version 3.0 replaced `sys.stdin` with an object which will raise an exception when any attempt is made to read from standard input or otherwise manipulate or reference the object:

```
IOError: sys.stdin access restricted by mod_wsgi
```

This restriction on standard input will however prevent the use of interactive debuggers for Python such as `pdb`. It can also interfere with Python modules which use the `isatty()` method of `sys.stdin` to determine whether an application is being run within an interactive session.

If it is required to be able to run such debuggers or other code which requires interactive input, the restriction on using standard input can be removed using a configuration directive:

```
WSGIRestrictStdin Off
```

This configuration directive must appear at global scope within the Apache configuration file outside of any VirtualHost container directives. It will remove the restriction on using standard input from all Python sub interpreters that mod_wsgi creates. There is no way using the configuration directive to remove the restriction from only one Python sub interpreter.

Note however that removing the restriction serves no purpose unless you also run the Apache web server in single process debug mode. This is because Apache normally makes use of multiple processes and would close standard input to prevent any process trying to read from standard input.

To run Apache in single process debug mode and thus allow an interactive Python debugger such as `pdb` to be used, your Apache instance should be shutdown and then the `httpd` program run explicitly:

```
httpd -X
```

For more details on using interactive debuggers in the context of mod_wsgi see documentation on *Debugging Techniques*.

Note that the restrictions on reading from stdin were removed in mod_wsgi 3.0 because it was found that people couldn't be bothered to fix their code. Instead they just used the documented workarounds, thereby propogating their

non portable WSGI application code. As such, since people just couldn't care, stopped promoting the idea of writing portable WSGI applications.

Registration Of Signal Handlers

Web servers upon which WSGI applications are hosted more often than not use signals to control their operation. The Apache web server in particular uses various signals to control its operation including the signals `SIGTERM`, `SIGINT`, `SIGHUP`, `SIGWINCH` and `SIGUSR1`.

If a WSGI application were to register their own signal handlers it is quite possible that they will interfere with the operation of the underlying web server, preventing it from being shutdown or restarted properly. As a general rule therefore, no WSGI application component should attempt to register its own signal handlers.

In order to actually enforce this, mod_wsgi will intercept all attempts to register signal handlers and cause the registration to be ignored. As warning that this is being done, a message will be logged to the Apache error log file of the form:

```
mod_wsgi (pid=123): Callback registration for signal 1 ignored.
```

If there is some very good reason that this feature should be disabled and signal handler registrations honoured, then the behaviour can be reversed using a configuration directive:

```
WSGIRestrictSignal Off
```

This configuration directive must appear at global scope within the Apache configuration file outside of any VirtualHost container directives. It will remove the restriction on signal handlers from all Python sub interpreters that mod_wsgi creates. There is no way using the configuration directive to remove the restriction from only one Python sub interpreter.

WSGI applications which are known to register conflicting signal handlers are CherryPy and TurboGears. If the ability to use signal handlers is reenabled when using these packages it prevents the shutdown and restart sequence of Apache from working properly and the main Apache process is forced to explicitly terminate the Apache child processes rather than waiting for them to perform an orderly shutdown. Similar issues will occur when using features of mod_wsgi daemon mode to recycle processes when a set number of requests has been reached or an inactivity timer has expired.

Pickling of Python Objects

The script files that mod_wsgi uses as the entry point for a WSGI application, although containing Python code, are not treated exactly the same as a Python code module. This has implications when it comes to using the 'pickle' module in conjunction which objects contained within the WSGI application script file.

In practice what this means is that neither function objects, class objects or instances of classes which are defined in a WSGI application script file should be stored using the "pickle" module.

In order to ensure that no strange problems at all are likely to occur, it is suggested that only basic builtin Python types, ie., scalars, tuples, lists and dictionaries, be stored using the "pickle" module from a WSGI application script file. That is, avoid any type of object which has user defined code associated with it.

The technical reasons for the limitations in the use of the "pickle" module in conjunction with WSGI application script files are further discussed in the document *Issues With Pickle Module*. Note that the limitations do not apply to standard Python modules and packages imported from within a WSGI application script file from directories on the standard Python module search path.

Expat Shared Library Conflicts

One of the Python modules which comes standard with Python is the 'pyexpat' module. This contains a Python wrapper for the popular 'expat' library. So as to avoid dependencies on third party packages the Python package actually contains a copy of the 'expat' library source code and embeds it within the 'pyexpat' module.

Prior to Python 2.5, there was however no attempt to properly namespace the public functions within the 'expat' library source code. The problem this causes with mod_wsgi is that Apache itself also provides its own copy of and makes use of the 'expat' library. Because the Apache version of the 'expat' library is loaded first, it will always be used in preference to the version contained with the Python 'pyexpat' module.

As a result, if the 'pyexpat' module is loaded into a WSGI application and the version of the 'expat' library included with Python is markedly different in some way to the Apache version, it can cause Apache to crash with a segmentation fault. It is thus important to ensure that Apache and Python use a compatible version of the 'expat' library to avoid this problem.

For further technical discussion of this issue and how to determine which version of the 'expat' library both Apache and Python use, see the document *Issues With Expat Library*.

MySQL Shared Library Conflicts

Shared library version conflicts can also occur with the MySQL client libraries. In this case the conflict is usually between an Apache module that uses MySQL directly such as mod_auth_mysql or mod_dbd_mysql, or an Apache module that indirectly uses MySQL such as PHP, and the Python 'MySQLdb' module. The result of conflicting library versions can be Apache crashing, or incorrect results beings returned from the MySQL client library for certain types of operations.

To ascertain if there is a conflict, you need to determine which versions of the shared library each package is attempting to use. This can be done by running, on Linux, the 'ldd' command to list the library dependencies. This should be done on any Apache modules that are being loaded, any PHP modules and the Python `_mysql` C extension module:

```
$ ldd /usr/lib/python2.3/site-packages/_mysql.so | grep mysql
    libmysqlclient_r.so.15 => /usr/lib/libmysqlclient_r.so.15 (0xb7d52000)

$ ldd /usr/lib/httpd/modules/mod_*.so | grep mysql
    libmysqlclient.so.12 => /usr/lib/libmysqlclient.so.12 (0xb7f00000)

$ ldd /usr/lib/php4/*.so | grep mysql
/usr/lib/php4/mysql.so:
    libmysqlclient.so.10 => /usr/lib/mysql/libmysqlclient.so.10 (0xb7f6e000)
```

If there is a difference in the version of the MySQL client library, or one version is reentrant and the other isn't, you will need to recompile one or both of the packages such that they use the same library.

SSL Shared Library Conflicts

When Apache is built, if it cannot find an existing SSL library that it can use or isn't told where one is that it should use, it will use a SSL library which comes with the Apache source code. When this SSL code is compiled it will be statically linked into the actual Apache executable. To determine if the SSL code is static rather than dynamically loaded from a shared library, on Linux, the 'ldd' command can be used to list the library dependencies. If an SSL library is listed, then code will not be statically compiled into Apache:

```
$ ldd /usr/local/apache/bin/httpd | grep ssl
    libssl.so.0.9.8 => /usr/lib/i686/cmov/libssl.so.0.9.8 (0xb79ab000)
```

Where a Python module now uses a SSL library, such as a database client library with SSL support, they would typically always obtain SSL code from a shared library. When however the SSL library functions have also been compiled statically into Apache, they can conflict and interfere with those from the SSL shared library being used by the Python module. Such conflicts can cause core dumps, or simply make it appear that SSL support in either Apache or the Python module is not working.

Python modules where this is known to cause a problem are, any database client modules which include support for connecting to the database using an SSL connection, and the Python 'hashlib' module introduced in Python 2.5.

In the case of the 'hashlib' module it will fail to load the internal C extension module called `_hashlib` because of the conflict. That `_hashlib` module couldn't be loaded is however not raised as an exception, and instead the code will fallback to attempting to load the older `_md5` module. In Python 2.5 however, this older `_md5` module is not generally compiled and so the following error will occur:

```
ImportError: No module named _md5
```

To resolve this problem it would be necessary to rebuild Apache and use the `--with-ssl` option to 'configure' to specify the location of the distinct SSL library that is being used by the Python modules.

Note that it has also been suggested that the !ImportError above can also be caused due to the 'python-hashlib' package not being installed. This might be the case on Linux systems where this module was separated from the main Python package.

Python MD5 Hash Module Conflict

Python provides in the form of the 'md5' module, routines for calculating MD5 message-digest fingerprint (checksum) values for arbitrary data. This module is often used in Python web frameworks for generating cookie values to be associated with client session information.

If a WSGI application uses this module, it is however possible that a conflict can arise if PHP is also being loaded into Apache. The end result of the conflict will be that the 'md5' module in Python can given incorrect results for hash values. For example, the same value may be returned no matter what the input data, or an incorrect or random value can be returned even for the same data. In the worst case scenario the process may crash.

As might be expected this can cause session based login schemes such as commonly employed by Python web frameworks such as Django, TurboGears or Trac to fail in strange ways.

The underlying trigger for all these problems appears to be a clash between the Python 'md5' module and the 'libmhash2' library used by the PHP 'mhash' module, or possibly also other PHP modules relying on md5 routines for cryptography such as the LDAP module for PHP.

This clash has come about because because md5 source code in Python was replaced with an alternate version when it was packaged for Debian. This version did not include in the "md5.h" header file some preprocessor defines to rename the md5 functions with a namespace prefix specific to Python:

```
#define MD5Init _PyDFSG_MD5Init
#define MD5Update _PyDFSG_MD5Update
#define MD5Final _PyDFSG_MD5Final
#define MD5Transform _PyDFSG_MD5Transform

void MD5Init(struct MD5Context *context);
void MD5Update(struct MD5Context *context, md5byte const *buf, unsigned len);
void MD5Final(unsigned char digest[16], struct MD5Context *context);
```

As a result, the symbols in the md5 module ended up being:

```
$ nm -D /usr/lib/python2.4/lib-dynload/md5.so | grep MD5
0000000000001b30 T MD5Final
```

```
0000000000001380 T MD5Init
00000000000013b0 T MD5Transform
0000000000001c10 T MD5Update
```

The symbols then clashed directly with the non namespaced symbols present in the 'libmhash2' library:

```
$ nm -D /usr/lib/libmhash.so.2 | grep MD5
00000000000069b0 T MD5Final
0000000000006200 T MD5Init
0000000000006230 T MD5Transform
0000000000006a80 T MD5Update
```

In Python 2.5 the md5 module is implemented in a different way and thus this problem should only occur with older versions of Python. For those older versions of Python, the only workaround for this problem at the present time is to disable the loading of the 'mhash' module or other PHP modules which use the 'libmhash2' library. This will avoid the problem with the Python 'md5' module, obviously however, not loading these modules into PHP may cause some PHP programs which rely on them to fail.

The actual cause of this problem having now been identified a patch has been produced and is recorded in Debian ticket:

- http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=440272

It isn't know when an updated Debian package for Python may be produced.

Python 'pysqlite' Symbol Conflict

Certain versions of 'pysqlite' module defined a global symbol 'cache_init'. This symbol clashes with a similarly named symbol present in the Apache mod_cache module. As a result of the clash, the two modules being loaded at the same time can cause the Apache process to crash or the following Python exception to be raised:

```
SystemError: NULL result without error in PyObject_Call
```

This problem is mentioned in pysqlite ticket:

- http://www.initd.org/tracker/pysqlite/ticket/174

and the release notes for version 2.3.3:

- http://www.initd.org/tracker/pysqlite/wiki/2.3.3_Changelog

of pysqlite To avoid the problem upgrade to pysqlite 2.3.3 or later.

Python Simplified GIL State API

In an attempt to simplify management of thread state objects when coding C extension modules for Python, Python 2.3 introduced the simplified API for GIL state management. Unfortunately, this API will only work if the code is running against the very first Python sub interpreter created when Python is initialised.

Because mod_wsgi by default assigns a Python sub interpreter to each WSGI application based on the virtual host and application mount point, code would normally never be executed within the context of the first Python sub interpreter created, instead a distinct Python sub interpreter would be used.

The consequences of attempting to use a C extension module for Python which is implemented against the simplified API for GIL state management in any sub interpreter besides the first, is that the code is likely to deadlock or crash the process. The only way around this issue is to ensure that any WSGI application which makes use of C extension modules which use this API, only runs in the very first Python sub interpreter created when Python is initialised.

To force a specific WSGI application to be run within the very first Python sub interpreter created when Python is initialised, the WSGIApplicationGroup directive should be used and the group set to '%{GLOBAL}':

```
WSGIApplicationGroup %{GLOBAL}
```

Extension modules for which this is known to be necessary are any which have been developed using SWIG and for which the `-threads` option was supplied to 'swig' when the bindings were generated. One example of this is the 'dbxml' module, a Python wrapper for the Berkeley Database, previously developed by !SleepyCat Software, but now managed by Oracle. Another package believed to have this problem in certain use cases is Xapian.

There is also a bit of a question mark over the Python Subversion bindings. This package also uses SWIG, however it is only some versions that appear to require that the very first sub interpreter created when Python is initialised be used. It is currently believed that this may be more to do with coding problems than with the `-threads` option being passed to the 'swig' command when the bindings were generated.

For all the affected packages, as described above it is believed though that they will work when application group is set to force the application to run in the first interpreter created by Python as described above.

Another option for packages which use SWIG generated bindings is not to use the `-threads` option when 'swig' is used to generate the bindings. This will avoid any problems and allow the package to be used in any sub interpreter. Do be aware though that by disabling thread support in SWIG bindings, that the GIL isn't released when C code is entered. The consequences of this are that if the C code blocks, the whole Python interpreter environment running in that process will be blocked, even requests being handled within other threads in different sub interpreters.

Reloading Python Interpreters

Note: The "Interpreter" reload mechanism has been removed in mod_wsgi version 2.0. This is because the problems with third party modules didn't make it a viable option. Its continued presence was simply complicating the addition of new features. As an alternative, daemon mode of mod_wsgi should be used and the "Process" reload mechanism added with mod_wsgi 2.0.

To make it possible to modify a WSGI application and have the whole application reloaded without restarting the Apache web server, mod_wsgi provides an interpreter reloading feature. This specific feature is enabled using the WSGIReloadMechanism directive, setting it to the value 'Interpreter' instead of its default value of 'Module':

```
WSGIReloadMechanism Interpreter
```

When this option is selected and script reloading is also enabled, when the WSGI application script file is modified, the next request which arrives will result in the Python sub interpreter which is hosting that WSGI application being destroyed. A new Python sub interpreter will then be created and the WSGI application reloaded including any changes made to normal Python modules.

For many WSGI applications this mechanism will generally work fine, however there are a few limitations on what is reloaded, plus some Python C extension modules can be incompatible with this feature.

The first issue is that although Python code modules will be destroyed and reloaded, because a C extension module is only loaded once and used across all Python sub interpreters for the life of the process, any changes to a C extension module will not be picked up.

The second issue is that some C extension modules may cache references to the Python interpreter object itself. Because there is no notification mechanism for letting a C extension module know when a sub interpreter is destroyed, it is possible that later on the C extension module may attempt to access the now destroyed Python interpreter. By this time the pointer reference is likely a dangling reference to unused memory or some completely different data and attempting to access or use it will likely cause the process to crash at some point.

A third issue is that the C extension module may cache references to Python objects in static variables but not actually increment the reference count on the objects in respect of its own reference to the objects. When the last Python sub interpreter to hold a reference to that Python object is destroyed, the object itself would be destroyed but the static

variable left with a dangling pointer. If a new Python sub interpreter is then created and the C extension module attempts to use that cached Python object, accessing it or using it will likely cause the process to crash at some point.

A few examples of Python modules which exhibit one or more of these problems are psycopg2, PyProtocols and lxml. In the case of !PyProtocols, because this module is used by TurboGears and sometimes used indirectly by Pylons applications, it means that the interpreter reloading mechanism can not be used with either of these packages. The reason for the problems with !PyProtocols appear to stem from its use of Pyrex generated code. The lxml package similarly uses Pyrex and is thus afflicted.

In general it is probably inadvisable to use the interpreter reload mechanism with any WSGI application which uses large or complicated C extension modules. It would be recommended for example that the interpreter reload mechanism not be used with Trac because of its use of the Python Subversion bindings. One would also need to be cautious if using any Python database client, although some success has been seen when using simple database adapters such as pysqlite.

Multiple Python Sub Interpreters

In addition to the requirements imposed by the Python GIL, other issues can also arise with C extension modules when multiple Python sub interpreters are being used. Typically these problems arise where an extension module caches a Python object from the sub interpreter which is initially used to load the module and then passes that object to code executing within secondary sub interpreters.

The prime example of where this would be a problem is where the code within the second sub interpreter attempts to execute a method of the Python object. When this occurs the result will be an attempt to execute Python code which doesn't belong to the current sub interpreter.

One result of this will be that if the code being executed attempts to import additional modules it will obtain those modules from the current sub interpreter rather than the interpreter the code belonged to. The result of this can be a unholy mixing of code and data owned by multiple sub interpreters leading to potential chaos at some point.

A more concrete outcome of such a mixing of code and data from multiple sub interpreters is where a file object from one sub interpreter is used within a different sub interpreter. In this sort of situation a Python exception will occur as Python will detect in certain cases that the object didn't belong to that interpreter:

```
exceptions.IOError: file() constructor not accessible in restricted mode
```

Problems with code being executed in restricted mode can also occur when the Python code and data marshalling features are used:

```
exceptions.RuntimeError: cannot unmarshal code objects in restricted execution mode
```

A further case is where the cached object is a class object and that object is used to create instances of that type of object for different sub interpreters. As above this can result in an unholy mixing of code and data from multiple sub interpreters, but at a more mundane level may become evident through the 'isinstance()' function failing when used to check the object instances against the local type object for that sub interpreter.

An example of a Python module which fails in this way is psycopg2, which caches an instance of the 'decimal.Decimal' type and uses it to create object instances for all sub interpreters. This particular problem in psycopg2 has been reported in psycopg2 ticket:

- http://www.initd.org/tracker/psycopg/ticket/192

and has been fixed in pyscopg2 source code. It isn't known however which version of psycopg2 this fix may have been released with. Another package believed to have this problem in certain use cases is lxml.

Because of the possibilty that extension module writers have not written their code to take into consideration it being used from multiple sub interpreters, the safest approach is to force all WSGI applications to run within the same application group, with that preferably being the first interpreter instance created by Python.

To force a specific WSGI application to be run within the very first Python sub interpreter created when Python is initialised, the WSGIApplicationGroup directive should be used and the group set to '%{GLOBAL}':

```
WSGIApplicationGroup %{GLOBAL}
```

If it is not feasible to force all WSGI applications to run in the same interpreter, then daemon mode of mod_wsgi should be used to assign different WSGI applications to their own daemon processes. Each would then be made to run in the first Python sub interpreter instance within their respective processes.

Memory Constrained VPS Systems

Virtual Private Server (VPS) systems typically always have constraints imposed on them in regard to the amount of memory or resources they are able to use. Various limits and related counts are described below:

Memory Limit Maximum virtual memory size a VPS/context can allocate.

Used Memory Virtual memory size used by a running VPS/context.

Max Total Memory Maximum virtual memory usage by VPS/context.

Context RSS Limit Maximum resident memory size a VPS/context can allocate. If limit is exceeded, VPS starts to use the host's SWAP.

Context RSS Resident memory size used by a running VPS/context.

Max RSS Memory Maximum resident memory usage by VPS/context.

Disk Limit Maximum disk space that can be used by VPS (calculated for the entire VPS file tree).

Used Disk Memory Disk space used by a VPS file tree.

Files Limit Maximum number of files that can be switched to a VPS/context.

Used Files Number of files used in a VPS/context.

TCP Sockets Limit Limit on the number of established connections in a virtual server.

Established Sockets Number of established connections in a virtual server.

In respect of the limits, when summary virtual memory size used by the VPS exceeds Memory Limit, processes can't allocate the required memory and will fail in unexpected ways. The general recommendation is that Context RSS Limit be set to be one third of Memory Limit.

Some VPS providers however appear to ignore such guidance, not perhaps understanding how virtual memory systems work, and set too restrictive a value on the Memory Limit of the VPS, to the extent that virtual memory use will exceed the Memory Limit even before actual memory use reaches Max RSS Memory or even perhaps before reaching Context RSS Limit.

This is especially a problem where the hosted operating system is Linux, as Linux uses a default per thread stack size which is excessive. When using Apache worker MPM with multiple threads, or mod_wsgi daemon mode and multiple worker threads, the amount of virtual memory quickly adds up causing the artificial Memory Limit to be exceeded.

Under Linux the default process stack size is 8MB. Where as other UNIX system typically use a much smaller per thread stack size in the order of 512KB, Linux inherits the process stack size and also uses it as the per thread stack size.

If running a VPS system and are having problems with Memory Limit being exceeded by the amount of virtual memory set aside by all applications running in the VPS, it will be necessary to override the default per thread stack size as used by Linux.

If you are using the Apache worker MPM, you will need to upgrade to Apache 2.2 if you are not already running it. Having done that you should then use the Apache directive !ThreadStackSize to lower the per thread stack size for threads created by Apache for the Apache child processes:

```
ThreadStackSize 524288
```

This should drop the amount of virtual memory being set aside by Apache for its child process and thus any WSGI application running under embedded mode.

If a WSGI application creates its own threads for performing background activities, it is also preferable that they also override the stack size set aside for that thread. For that you will need to be using at least Python 2.5. The WSGI application should be ammended to execute:

```
import thread
thread.stack_size(524288)
```

If using mod_wsgi daemon mode, you will need to use mod_wsgi 2.0 and override the per thread stack size using the 'stack-size' option to the WSGIDaemonProcess directive:

```
WSGIDaemonProcess example stack-size=524288
```

If you are unable to upgrade to Apache 2.2 and/or mod_wsgi 2.0, the only other option you have for affecting the amount of virtual memory set aside for the stack of each thread is to override the process stack size. If you are using a standard Apache distribution, this can be done by adding to the 'envvars' file for the Apache installation:

```
ulimit -s 512
```

If using a customised Apache installation, such as on RedHat, the 'envvars' file may not exist. In this case you would need to add this into the actual startup script for Apache. For RedHat this is '/etc/sysconfig/httpd'.

Note that although 512KB is given here as an example, you may in practice need to adjust this higher if you are using third party C extension modules for Python which allocate significant amounts of memory on the stack.

OpenBSD And Thread Stack Size

When using Linux the excessive amount of virtual memory set aside for the stack of each thread can cause problems in memory constrained VPS systems. Under OpenBSD the opposite problem can occur in that the default per thread stack size can be too small. In this situation the same mechanisms as used above for adjusting the amount of virtual memory set aside can be used, but in this case to increase the amount of memory to be greater than the default value.

Although it has been reported that the default per thread stack size on OpenBSD can be a problem, it isn't known what it defaults too and thus whether it is too low, or whether it was just the users specific application which was attempting to allocate too much memory from the stack.

Python Oracle Wrappers

When using WSGIDaemonProcess directive, it is possible to use the 'display-name' option to set what the name of the process is that will be displayed in output from BSD derived 'ps' programs and some other monitoring programs. This allows one to distinguish the WSGI daemon processes in a process group from the normal Apache 'httpd' processes.

The mod_wsgi package accepts the magic string '%{GROUP}' as value to the WSGIDaemonProcess directive to indicate that mod_wsgi should construct the name of the processes based on the name of the process group. Specifically, if you have:

```
WSGIDaemonprocess mygroup display-name=%{GROUP}
```

then the name of the processes in that process group would be set to the value:

```
(wsgi:mygroup)
```

This generally works fine, however causes a problem when the WSGI application makes use of the 'cx_Oracle' module for wrapping Oracle client libraries in Python. Specifically, Oracle client libraries can produce the error:

```
ORA-06413: Connection not open.
```

This appears to be caused by the use of brackets, ie., '()' in the name of the process. It is therefore recommended that you explicitly provide the name to use for the process and avoid these characters and potentially any non alphanumeric characters to be extra safe.

This issue is briefly mentioned in:

- http://www.dba-oracle.com/t_ora_06413_connection_not_open.htm

Non Blocking Module Imports

In Python 2.6 non blocking module imports were added as part of the Python C API in the form of the function PyImport_ImportModuleNoBlock(). This function was introduced to prevent deadlocks with module imports in certain circumstances. Unfortunately, for valid reasons or not, use of this function has been sprinkled through Python standard library modules as well as third party modules.

Although the function may have been created to fix some underlying issue, its usage has caused a new set of problems for multithreaded programs which defer module importing until after threads have been created. With mod_wsgi this is actually the norm as the default mode of operation is that code is lazily loaded only when the first request arrives which requires it.

A classic example of the sorts of problems use of this function causes is the error:

```
ImportError: Failed to import _strptime because the import lock is held by another␣
↪thread.
```

This particular error occurs when 'time.strptime()' is called for the first time and it so happens that another thread is in the process of doing a module import and holds the global module import lock.

It is believed that the fact this can happen indicates that Python is flawed in using the PyImport_ImportModuleNoBlock(). Unfortunately, when this issue has been highlighted in the past, people seemed to think it was acceptable and the only solution, rather than fixing the Python standard library, was to ensure that all module imports are done before any threads are created.

This response is frankly crazy and you can expect all manner of random problems related to this to crop up as more and more people start using the PyImport_ImportModuleNoBlock() function without realising that it is a really bad idea in the context of a multithreaded system.

Although no hope is held out for the issue being fixed in Python, a problem report has still been lodged and can be found at:

```
* http://bugs.python.org/issue8098
```

The only work around for the problem is to ensure that all module imports related to modules on which the PyImport_ImportModuleNoBlock() function is used be done explicitly or indirectly when the WSGI script file is loaded. Thus, to get around the specific case above, add the following into the WSGI script file:

```
import _strptime
```

There is nothing that can be done in mod_wsgi to fix this properly as the set of modules that might have to be forceably imported is unknown. Having a hack to import them just to avoid the problem is also going to result in unnecessary memory usage if the application didn't actually need them.

Frequently Asked Questions

Apache Process Crashes

Q: Why when the mod_wsgi module is initially being loaded by Apache, do the Apache server processes crash with a 'segmentation fault'?

A: This is nearly always caused due to mod_python also being loaded by Apache at the same time as mod_wsgi and the Python installation not providing a shared library, or mod_python having originally being built against a static Python library. This is especially a problem with older Linux distributions before they started shipping with Python as a shared library.

Further information on these problems can be found in various sections of [InstallationIssues Installation Issues].

Q: Why when first request is made against a WSGI application does the Apache server process handling the request crash with a 'segmentation fault'?

A: This is nearly always caused due to a shared library version conflict. That is, Apache or some Apache module is linked against a different version of a library than that which is being used by a particular Python module that the WSGI application makes use of. The most common culprits are the expat and MySQL libraries, but it can also occur with other shared libraries.

Another cause of a process crash only upon the first request can be a third party C extension module for Python which has not been implemented so as to work within a secondary Python sub interpreter. The Python bindings for Subversion are a particular example, with the Python module only working correctly if the WSGI application is forced to run within the first interpreter instance created by Python.

Further information on these problems can be found in various sections of *Application Issues*. The problems with the expat library are also gone into in more detail in *Issues With Expat Library*.

Q: Why am I seeing the error message 'premature end of script headers' in the Apache error logs.

A: If using daemon mode, this is a symptom of the mod_wsgi daemon process crashing when handling a request. You would probably also see the message 'segmentation fault'. See answer for question about 'segmentation fault' above.

This error message can also occur where you haven't configured Apache correctly and your WSGI script file is being executed as a CGI script instead.

HTTP Error Responses

Q: When I try to use mod_wsgi daemon mode I get the error response '503 Service Temporarily Unavailable'.

A: The standard Apache runtime directory has restricted access and the Apache child process cannot access the daemon process sockets. You will need to use the WSGISocketPrefix directive to specify an alternative location for storing of runtime files such as sockets.

For further information see section 'Location Of UNIX Sockets' of [ConfigurationIssues Configuration Issues].

Q: I am getting a HTTP 500 error response and I can't find any error in the Apache error logs.

A: Some users of mod_wsgi 1.3/2.0 and older minor revisions, are finding that mod_wsgi error messages are going missing, or ending up in the main Apache error log file rather than a virtual host specific error log file. Specifically, this is occurring when Apache ErrorLog directive is being used inside of a VirtualHost container.

It is not known exactly what operating system setup and/or Apache configuration is the trigger for this problem. To avoid the problem, use a newer version of mod_wsgi.

HTTP Error Log Messages

Q: Why do I get the error 'IOError: client connection closed' appearing in the error logs?

A: This occurs when the HTTP client making the request closes the connection before the complete response for a request has been written.

This can occur where a user force reloads a web page before it had been completely displayed. It can also occur when using benchmarking tools such as 'ab' as they will over commit on the number of requests they make when doing concurrent requests, killing off any extra requests once the required number has been reached.

In general this error message can be ignored.

Application Reloading

Q: Do I have to restart Apache every time I make a change to the Python code for my WSGI application?

A: If your WSGI application is contained totally within the WSGI script file and it is that file that you are changing, then no you don't. In this case the WSGI script file will be automatically reloaded when a change is made provided that script reloading hasn't been disabled.

If the code you are changing lies outside of the WSGI script file then what you may need to do will depend on how mod_wsgi is being used.

If embedded mode of mod_wsgi is being used, the only option is to restart Apache. You could set Apache configuration directive MaxRequestsPerChild to 1 to force a reload of the application on every request, but this is not recommended because it will perform as bad as or as worse as CGI and will also affect serving up of static files and other applications being hosted by the same Apache instance.

If using daemon mode with a single process you can send a SIGINT signal to the daemon process using the 'kill' command, or have the application send the signal to itself when a specific URL is triggered.

If using daemon mode, with any number of processes, and the process reload mechanism of mod_wsgi 2.0 has been enabled, then all you need to do is touch the WSGI script file, thereby updating its modification time, and the daemon processes will automatically shutdown and restart the next time they receive a request.

Use of daemon mode and the process reload mechanism is the preferred mechanism for handling automatic reloading of code after changes.

More details on how source code reloading works with mod_wsgi can be found in *Reloading Source Code*.

Q: Why do requests against my application seem to take forever, but then after a bit they all run much quicker?

A: This is because mod_wsgi by default performs lazy loading of any application. That is, an application is only loaded the first time that a request arrives which targets that WSGI application. This means that those initial requests will incur the overhead of loading all the application code and performing any startup initialisation.

This startup overhead can appear to be quite significant, especially if using Apache prefork MPM and embedded mode. This is because the startup cost is incurred for each process and with prefork MPM there are typically a lot more processes that if using worker MPM or mod_wsgi daemon mode. Thus, as many requests as there are processes will run slowly and everything will only run full speed once code has all been loaded.

Note that if recycling of Apache child processes or mod_wsgi daemon processes after a set number of requests is enabled, or for embedded mode Apache decides itself to reap any of the child processes, then you can periodically see these delayed requests occurring.

Some number of the benchmarks for mod_wsgi which have been posted do not take into mind these start up costs and wrongly try to compare the results to other systems such as fastcgi or proxy based systems where the application code would be preloaded by default. As a result mod_wsgi is painted in a worse light than is reality. If mod_wsgi is configured correctly the results would be better than is shown by those benchmarks.

For some cases, such as when WSGIScriptAlias is being used, it is actually possible to preload the application code when the processes first starts, rather than when the first request arrives. To preload an application see the WSGIImportScript directive.

By preloading the application code you would not normally see delays in requests being handled. The only exception to this would be when running a single process under mod_wsgi daemon mode and the process is being restarted when a maximum number of requests arrives or explicitly via one of the means to trigger reloading of application code. Delays here can be avoided by running at least two processes in the daemon process group. This is because when one process is restarting, the others can handle the requests.

Execution Environment

Q: Why do I get the error 'IOError: sys.stdout access restricted by mod_wsgi'?

A: A portable WSGI application or application component should not output anything to standard output. This is because some WSGI hosting mechanisms use standard output to communicate with the web server. If a WSGI application outputs anything to standard output it will thus potentially interleave with the response sent back to the client.

To promote portability of WSGI applications, mod_wsgi by default restricts direct use of 'sys.stdout' and 'sys.stdin'. Because the 'print' statement defaults to outputing text to 'sys.stdout', using 'print' for debugging purposes can cause this error.

For more details about this issue, including how applications should do logging and how to disable this restriction see section 'Writing To Standard Output' in *Application Issues* and section 'Apache Error Log Files' in *Debugging Techniques*.

Q: Can mod_wsgi be used with Python virtual environments created using Ian Bicking's 'virtualenv' package?

A: Yes. For more details see *Virtual Environments*.

Access Control Mechanisms

Q: Why are client user credentials not being passed through to the WSGI application in the 'HTTP_AUTHORIZATION' variable of the WSGI environment?

A: User credentials are not passed by default as doing so is insecure and could expose a users password to WSGI applications which shouldn't be permitted to see it. Such a situation might occur within a corporate setting where HTTP authentication mechanisms were used to control access to a corporate web server but it was possible for users to provide their own web pages. The last thing a system administator will want is normal users being able to see other users passwords.

As a result, the passing of HTTP authentication credentials must be explicitly enabled by the web server administrator. This can only be done using directives placed in the main Apache confguration file.

For further information see *Access Control Mechanisms* and the documentation for the WSGIPassAuthorization directive.

Q: Is there a way of having a WSGI application provide user authentication for resources outside of the application such as static files, CGI scripts or even a distinct application. In other words, something akin to being able to define access, authentication and authorisation handlers in mod_python?

A: Providing you are using Apache 2.0 or later, version 2.0 of mod_wsgi provides support for hooking into the Apache access, authentication and authorisation handler phases. This doesn't allow full control of how the Apache handler is implemented, but does allow control over how user credentials are validated, determination of what groups a user is a member of and whether specific hosts are allowed access. This is generally more than sufficient and makes the task somewhat simpler than needing to implement a full handler like in mod_python as Apache and mod_wsgi do all the hard work.

For further information see *Access Control Mechanisms*.

Checking Your Installation

When debugging mod_wsgi or a WSGI application, it is import to be able to understand how mod_wsgi has been installed, what Apache and/or Python it uses and how those systems have been configured, plus under what configuration the WSGI application is running.

This document details various such checks that can be made. The primary purpose of providing this information is so that when people ask questions on the mod_wsgi mailing list, they can be directed here to perform certain checks as a way of collecting additional information needed to help debug their problem.

If you are reading this document because you have been directed here from the mailing list, then ensure that you actually provide the full amount of detail obtained from the checks and not leave out information. When you leave out information then it means guesses have to be made about your setup which makes it harder to debug your problems.

Apache Build Information

Information related to what version of Apache is being used and how it is built is obtained in a number of ways. The primary means is from the Apache 'httpd' executable itself using command line options. The main such option is the `-V` option.

On most systems the standard Apache executable supplied with the operating system is located at '/usr/sbin/httpd'. On MacOS X, for the operating system supplied Apache the output from this is:

```
$ /usr/sbin/httpd -V
Server version: Apache/2.2.14 (Unix)
Server built:   Feb 10 2010 22:22:39
Server's Module Magic Number: 20051115:23
Server loaded:  APR 1.3.8, APR-Util 1.3.9
Compiled using: APR 1.3.8, APR-Util 1.3.9
Architecture:   64-bit
Server MPM:     Prefork
  threaded:     no
    forked:     yes (variable process count)
Server compiled with....
 -D APACHE_MPM_DIR="server/mpm/prefork"
 -D APR_HAS_SENDFILE
 -D APR_HAS_MMAP
 -D APR_HAVE_IPV6 (IPv4-mapped addresses enabled)
 -D APR_USE_FLOCK_SERIALIZE
 -D APR_USE_PTHREAD_SERIALIZE
 -D SINGLE_LISTEN_UNSERIALIZED_ACCEPT
 -D APR_HAS_OTHER_CHILD
 -D AP_HAVE_RELIABLE_PIPED_LOGS
 -D DYNAMIC_MODULE_LIMIT=128
 -D HTTPD_ROOT="/usr"
 -D SUEXEC_BIN="/usr/bin/suexec"
```

```
-D DEFAULT_PIDLOG="/private/var/run/httpd.pid"
-D DEFAULT_SCOREBOARD="logs/apache_runtime_status"
-D DEFAULT_LOCKFILE="/private/var/run/accept.lock"
-D DEFAULT_ERRORLOG="logs/error_log"
-D AP_TYPES_CONFIG_FILE="/private/etc/apache2/mime.types"
-D SERVER_CONFIG_FILE="/private/etc/apache2/httpd.conf"
```

The most important details here are:

- The version of Apache from the 'Server version' entry.
- The MPM which Apache has been compiled to use from the 'Server MPM' entry.

Although this has a section which appears to indicate what preprocessor options the server was compiled with, it is a massaged list. What is often more useful is the actual arguments which were supplied to the 'configure' command when Apache was built.

To determine this information you need to do the following.

- Work out where 'apxs2' or 'apxs' is installed.
- Open this file and find setting for '$installbuilddir'.
- Open the 'config.nice' file in the directory specified for build directory.

On MacOS X, for the operating system supplied Apache this file is located at '/usr/share/httpd/build/config.nice'. The contents of the file is:

```
#! /bin/sh
#
# Created by configure

"/SourceCache/apache/apache-747.1/httpd/configure" \
"--prefix=/usr" \
"--enable-layout=Darwin" \
"--with-apr=/usr" \
"--with-apr-util=/usr" \
"--with-pcre=/usr/local/bin/pcre-config" \
"--enable-mods-shared=all" \
"--enable-ssl" \
"--enable-cache" \
"--enable-mem-cache" \
"--enable-proxy-balancer" \
"--enable-proxy" \
"--enable-proxy-http" \
"--enable-disk-cache" \
"$@"
```

Not only does this indicate what features of Apache have been compiled in, it also indicates by way of the `--enable-layout` option what custom Apache installation layout has been used.

Apache Modules Loaded

Modules can be loaded into Apache statically, or can be loaded dynamically at run time based on Apache configuration files.

If modules have been statically compiled into Apache, usually it would be evident by what 'configure' arguments have been used when Apache was built. To verify what exactly what is compiled in statically, you can use the `-l` option to the Apache executable.

On MacOS X, for the operating system supplied Apache the output from running `-l` option is:

```
$ /usr/sbin/httpd -l
Compiled in modules:
  core.c
  prefork.c
  http_core.c
  mod_so.c
```

This indicates that the only module that is loaded statically is 'mod_so'. This is actually the Apache module that handles the task of dynamically loading other Apache modules.

For a specific Apache configuration, you can determine what Apache modules will be loaded dynamically by using the `-M` option for the Apache executable.

On MacOS X, for the operating system supplied Apache the output from running `-M` option, where the only additional module added is mod_wsgi, is:

```
$ /usr/sbin/httpd -M
Loaded Modules:
 core_module (static)
 mpm_prefork_module (static)
 http_module (static)
 so_module (static)
 authn_file_module (shared)
 authn_dbm_module (shared)
 authn_anon_module (shared)
 authn_dbd_module (shared)
 authn_default_module (shared)
 authz_host_module (shared)
 authz_groupfile_module (shared)
 authz_user_module (shared)
 authz_dbm_module (shared)
 authz_owner_module (shared)
 authz_default_module (shared)
 auth_basic_module (shared)
 auth_digest_module (shared)
 cache_module (shared)
 disk_cache_module (shared)
 mem_cache_module (shared)
 dbd_module (shared)
 dumpio_module (shared)
 ext_filter_module (shared)
 include_module (shared)
 filter_module (shared)
 substitute_module (shared)
 deflate_module (shared)
 log_config_module (shared)
 log_forensic_module (shared)
 logio_module (shared)
 env_module (shared)
 mime_magic_module (shared)
 cern_meta_module (shared)
 expires_module (shared)
 headers_module (shared)
 ident_module (shared)
 usertrack_module (shared)
 setenvif_module (shared)
 version_module (shared)
```

```
 proxy_module (shared)
 proxy_connect_module (shared)
 proxy_ftp_module (shared)
 proxy_http_module (shared)
 proxy_ajp_module (shared)
 proxy_balancer_module (shared)
 ssl_module (shared)
 mime_module (shared)
 dav_module (shared)
 status_module (shared)
 autoindex_module (shared)
 asis_module (shared)
 info_module (shared)
 cgi_module (shared)
 dav_fs_module (shared)
 vhost_alias_module (shared)
 negotiation_module (shared)
 dir_module (shared)
 imagemap_module (shared)
 actions_module (shared)
 speling_module (shared)
 userdir_module (shared)
 alias_module (shared)
 rewrite_module (shared)
 bonjour_module (shared)
 wsgi_module (shared)
Syntax OK
```

The names reflect that which would have been used with the LoadModule line in the Apache configuration and not the name of the module file itself.

The order in which modules are listed can be important in some cases where a module doesn't explicitly designate in what order a handler should be applied relative to other Apache modules.

Global Accept Mutex

Because Apache is a multi process server, it needs to use a global cross process mutex to control which of the Apache child processes get the next chance to accept a connection from a HTTP client.

This cross process mutex can be implemented using a variety of different mechanisms and exactly which is used can vary based on the operating system. Which mechanism is used can also be overridden in the Apache configuration if absolutely required.

A simlar instance of a cross process mutex is also used for each mod_wsgi daemon process group to mediate which process in the daemon process group gets to accept the next request proxied to that daemon process group via the Apache child processes.

The list of mechanisms which might be used to implement the cross process mutex are as follows:

- flock
- fcntl
- sysvsem
- posixsem
- pthread

In the event that there are issues which communicating between the Apache child processes and the mod_wsgi daemon process in particular, it can be useful to know what mechanism is used to implement the cross process mutex.

By default, the Apache configuration files would not specify a specific mechanism, and instead which is used would be automatically selected by the underlying Apache runtime libraries based on various build time and system checks about what is the prefered mechanism for a particular operating system.

Which mechanism is used by default can be determined from the build information displayed by the `-V` option to the Apache executable described previously. The particular entries of interest are those with 'SERIALIZE' in the name of the macro.

On MacOS X, using operating system supplied Apache, the entries of interest are:

```
-D APR_USE_FLOCK_SERIALIZE
-D APR_USE_PTHREAD_SERIALIZE
```

As the entries are used in order, what this indicates is that Apache will by default use the 'flock' mechanism to implement the cross process mutex.

In comparison, on a Linux system, the entries of interest may be:

```
-D APR_USE_SYSVSEM_SERIALIZE
-D APR_USE_PTHREAD_SERIALIZE
```

which indicates that 'sysvsem' mechanism is instead used.

This mechanism is also what would be used by mod_wsgi by default as well for the cross process mutex for daemon process groups.

This mechanism will be different where the AcceptMutex and WSGIAcceptMutex directives are used.

If the AcceptMutex directive is defined in the Apache configuration file, then what ever mechanism is specified will be used instead for Apache child processes. Provided that Apache 2.2 or older is used, and WSGIAcceptMutex is not specified, then when AcceptMutex is used, that will also then be used by mod_wsgi daemon processes as well.

In the case of Apache 2.4 and later, AcceptMutex will no longer override the default for mod_wsgi daemon process groups, and instead WSGIAcceptMutex must be specified seperately if it needs to be overridden for both.

Either way, you should check the Apache configuration files as to whether either AcceptMutex or WSGIAcceptMutex directives are used as they will override the defaults calculated above. Under normal circumstances neither should be set as default would always be used.

If wanting to look at overriding the default mechanism, what options exist for what mechanism can be used will be dependent on the operating system being used. There are a couple of ways this can be determined.

The first is to find the 'apr.h' header file from the Apache runtime library installation that Apache was compiled against. In that you will find entries similar to the 'USE' macros above. You will also find 'HAS' entries. In this case we are interested in the 'HAS' entries.

On MacOS X, with the operating system supplied APR library, the entries in 'apr.h' are:

```
#define APR_HAS_FLOCK_SERIALIZE           1
#define APR_HAS_SYSVSEM_SERIALIZE         1
#define APR_HAS_POSIXSEM_SERIALIZE        1
#define APR_HAS_FCNTL_SERIALIZE           1
#define APR_HAS_PROC_PTHREAD_SERIALIZE    0
```

The available mechanisms are those defined to be '1'.

Finding where the right 'apr.h' is located may be tricky, so an easier way is to trick Apache into generating an error message to list what the available mechanisms are. To do this, in turn, add entries into the Apache configuration files, at global scope of:

```
AcceptMutex xxx
```

and:

```
WSGIAcceptMutex xxx
```

For each run the `-t` option on the Apache program executable.

On MacOS X, with the operating system supplied APR library, this yields:

```
$ /usr/sbin/httpd -t
Syntax error on line 501 of /private/etc/apache2/httpd.conf:
xxx is an invalid mutex mechanism; Valid accept mutexes for this platform \
 and MPM are: default, flock, fcntl, sysvsem, posixsem.
```

for AcceptMutex and for WSGIAcceptMutex:

```
$ /usr/sbin/httpd -t
Syntax error on line 501 of /private/etc/apache2/httpd.conf:
Accept mutex lock mechanism 'xxx' is invalid. Valid accept mutex mechanisms \
 for this platform are: default, flock, fcntl, sysvsem, posixsem.
```

The list of available mechanisms should normally be the same in both cases.

Using the value of 'default' indicates that which mechanism is used is left up to the APR library.

Python Shared Library

When mod_wsgi is built, the 'mod_wsgi.so' file should be linked against Python via a shared library. If it isn't and it is linked against a static library, various issues can arise. These include additional memory usage, plus conflicts with mod_python if it is also loaded in same Apache.

To validate that 'mod_wsgi.so' is using a shared library for Python, on most UNIX systems the 'ldd' command is used. For example:

```
$ ldd mod_wsgi.so
 linux-vdso.so.1 =>  (0x00007fffeb3fe000)
 libpython2.5.so.1.0 => /usr/local/lib/libpython2.5.so.1.0 (0x00002adebf94d000)
 libpthread.so.0 => /lib/libpthread.so.0 (0x00002adebfcba000)
 libdl.so.2 => /lib/libdl.so.2 (0x00002adebfed6000)
 libutil.so.1 => /lib/libutil.so.1 (0x00002adec00da000)
 libc.so.6 => /lib/libc.so.6 (0x00002adec02dd000)
 libm.so.6 => /lib/libm.so.6 (0x00002adec0635000)
 /lib64/ld-linux-x86-64.so.2 (0x0000555555554000)
```

What you want to see is a reference to an instance of 'libpythonX.Y.so'. Normally the operating system shared library version suffix would always be '1.0'. What it is shouldn't really matter though.

This reference should refer to the actual Python shared library for your Python installation.

Do note though, that 'ldd' will take into consideration any local user setting of the 'LD_LIBRARY_PATH' environment variable. That is, 'ldd' will also search any directories listed in that environment variable for shared libraries.

Although that environment variable may be defined in your user account, it will not normally be defined in the environment of the account that Apache starts up as. Thus, it is important that you unset the 'LD_LIBRARY_PATH' environment variable when running 'ldd'.

If you run the check with and without 'LD_LIBRARY_PATH' set and find that without it that a different, or no Python shared library is found, then you will likely have a problem. For the case of it not being found, Apache will fail to

start. For where it is found but it is a different installation to that which you want used, subtle problems could occur due to C extension modules for Python being used which were compiled against that installation.

For example, if 'LD_LIBRARY_PATH' contained the directory '/usr/local/lib' and you obtained the results above, but when you unset it, it picked up shared library from '/usr/lib' instead, then you may end up with problems if for a different installation. In this case you would see:

```
$ unset LD_LIBRARY_PATH
$ ldd mod_wsgi.so
 linux-vdso.so.1 =>  (0x00007fffeb3fe000)
 libpython2.5.so.1.0 => /usr/lib/libpython2.5.so.1.0 (0x00002adebf94d000)
 libpthread.so.0 => /lib/libpthread.so.0 (0x00002adebfcba000)
 libdl.so.2 => /lib/libdl.so.2 (0x00002adebfed6000)
 libutil.so.1 => /lib/libutil.so.1 (0x00002adec00da000)
 libc.so.6 => /lib/libc.so.6 (0x00002adec02dd000)
 libm.so.6 => /lib/libm.so.6 (0x00002adec0635000)
 /lib64/ld-linux-x86-64.so.2 (0x0000555555554000)
```

Similarly, if not found at all, you would see:

```
$ unset LD_LIBRARY_PATH
$ ldd mod_wsgi.so
 linux-vdso.so.1 =>  (0x00007fffeb3fe000)
 libpython2.5.so.1.0 => not found
 libpthread.so.0 => /lib/libpthread.so.0 (0x00002adebfcba000)
 libdl.so.2 => /lib/libdl.so.2 (0x00002adebfed6000)
 libutil.so.1 => /lib/libutil.so.1 (0x00002adec00da000)
 libc.so.6 => /lib/libc.so.6 (0x00002adec02dd000)
 libm.so.6 => /lib/libm.so.6 (0x00002adec0635000)
 /lib64/ld-linux-x86-64.so.2 (0x0000555555554000)
```

If you have this problem, then it would be necessary to set 'LD_RUN_PATH' environment variable to include directory containing where Python library resides when building mod_wsgi, or set 'LD_LIBRARY_PATH' in startup file for Apache such that it is also set for Apache when run. For standard Apache installation the latter would be done in 'envvars' file in same directory as Apache program executable. For some Linux installations would need to be done in init scripts for Apache.

Note that MacOS X doesn't use 'LD_LIBRARY_PATH' nor have 'ldd'. On MacOS X, instead of 'ldd' you can use 'otool -L':

```
$ otool -L mod_wsgi.so
mod_wsgi.so:
  /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 125.2.0)
  /System/Library/Frameworks/Python.framework/Versions/2.6/Python (compatibility␣
↪version 2.6.0, current version 2.6.1)
```

If using standard MacOS X compilers and not using Fink or !MacPorts, there generally should not ever be any issues with whether it is a shared library or not as everything should just work.

The only issue with MacOS X is that for whatever reason, the location dependency for the shared library (framework) isn't always encoded into 'mod_wsgi.so' correctly. This seems to vary between what Python installation was used and what MacOS X operating system version. In this case, if multiple installations of same version of Python in different locations, may find the system installation rather than your custom installation.

In that situation you may need to use the `--disable-framework` option to 'configure' script for mod_wsgi. This doesn't actually disable use of the framework, but does change how it links to use a more traditional library style linking rather than framework linking. This seems to resolve the problems in most cases.

Python Installation In Use

Although the 'mod_wsgi.so' file may be finding a specific Python shared library and that may be from the correct installation, the Python library when initialised doesn't actually know from where it came. As such, it uses a series of checks to try and determine where the Python installation is actually located.

This check has various subtleties and how it works varies depending on the platform used. At its simplest though, on most UNIX systems it will check all directories listed in the 'PATH' environment variable of the process. In each of those directories it will look for the 'python' program. When it finds such a file, it will then look for a corresponding 'lib' directory containing a valid Python installation for the same version of Python as is being run.

When it finds such a directory, the home for the Python installation will be taken as being the parent directory of the directory containing the 'python' program file found.

Because this search is dependent on the 'PATH' environment variable, which is likely set to a minimal set of directories for the Apache user, then if you are using a Python installation in a non standard location, then it may not properly find the location of that installation.

The easiest way to validate which Python installation is being used is to use a test WSGI script to output the value of 'sys.prefix':

```
import sys

def application(environ, start_response):
    status = '200 OK'

    output = u''
    output += u'sys.version = %s\n' % repr(sys.version)
    output += u'sys.prefix = %s\n' % repr(sys.prefix)

    response_headers = [('Content-type', 'text/plain'),
                        ('Content-Length', str(len(output)))]
    start_response(status, response_headers)

    return [output.encode('UTF-8')]
```

For standard Python installation on a Linux system, this would produce something like:

```
sys.version = "'2.6.1 (r261:67515, Feb 11 2010, 00:51:29) \\n[GCC 4.2.1 (Apple Inc.␣
↪build 5646)]'"
sys.prefix = '/usr'
```

Thus, if you were expecting to pick up a separate Python installation located under '/usr/local' or elsewhere, this would be indicative of a problem.

It can also be worthwhile to check that the Python module search path also looks correct. This can be done by using a test WSGI script to output the value of 'sys.path':

```
import sys

def application(environ, start_response):
    status = '200 OK'
    output = u'sys.path = %s' % repr(sys.path)

    response_headers = [('Content-type', 'text/plain'),
                        ('Content-Length', str(len(output)))]
    start_response(status, response_headers)

    return [output.encode('UTF-8')]
```

In both cases, even if incorrect location is being used for Python installation and even if there is no actual Python installation of the correct version under that root directory, then these test scripts should still run as 'sys' module is a builtin module which can be satisified via just the Python library.

If debugging, whether there is a Python installation underneath that root directory, the subdirectory which you would want to look for is 'lib/pythonX.Y' corresponding to version of Python being used.

If the calculated directory is wrong, then you will need to use the WSGIPythonHome directory to set the location to the correct value. The value to use is what 'sys.prefix' is set to when the correct Python is run from the command line and 'sys.prefix' output:

```
>>> import sys
>>> print sys.prefix
/usr/local
```

Thus for case where installed under '/usr/local', would use:

```
WSGIPythonHome /usr/local
```

Embedded Or Daemon Mode

WSGI applications can run in either embedded mode or daemon mode. In the case of embedded mode, the WSGI application runs within the Apache child processes themselves. In the case of daemon mode, they run within a separate set of processes managed by mod_wsgi.

To determine what mode a WSGI application is running under, replace its WSGI script with the test WSGI script as follows:

```
import sys

def application(environ, start_response):
    status = '200 OK'
    output = u'mod_wsgi.process_group = %s' % repr(environ['mod_wsgi.process_group'])
    response_headers = [('Content-type', 'text/plain'),
                        ('Content-Length', str(len(output)))]
    start_response(status, response_headers)

    return [output.encode('UTF-8')]
```

If the configuration is such that the WSGI application is running in embedded mode, then you will see:

```
mod_wsgi.process_group = ''
```

This actually corresponds to the directive:

```
WSGIProcessGroup %{GLOBAL}
```

having being used, or the same value being used to the 'process-group' directive of WSGIScriptAlias. Do note though that these are also actually the defaults for these if not explicitly defined.

If the WSGI application is actually running in daemon mode, then a non empty string will instead be shown corresponding to the name of the daemon process group used.

Sub Interpreter Being Used

As well as WSGI application being able to be delegated to run in either embedded mode or daemon mode, within the process it ends up running in, it can be delegated to a specific Python sub interpreter.

To determine which Python sub interpreter is being used within the process the WSGI application is being run use the test WSGI script of:

```
import sys

def application(environ, start_response):
    status = '200 OK'
    output = u'mod_wsgi.application_group = %s' % repr(environ['mod_wsgi.application_
↪group'])

    response_headers = [('Content-type', 'text/plain'),
                        ('Content-Length', str(len(output)))]
    start_response(status, response_headers)

    return [output.encode('UTF-8')]
```

If being run in the main interpreter, ie., the first interpreter created by Python, this will output:

```
mod_wsgi.application_group = ''
```

This actually corresponds to the directive:

```
WSGIApplicationGroup %{GLOBAL}
```

having being used, or the same value being used to the 'application-group' directive of WSGIScriptAlias.

The default for these if not defined is actually '%{RESOURCE}'. This will be a value made up from the name of the virtual host or server, the port on which connection was accepted and the mount point of the WSGI application. The port however is actually dropped where port is 80 or 443.

An example of what you would expect to see is:

```
mod_wsgi.application_group = 'tests.example.com|/interpreter.wsgi'
```

This corresponds to server name of 'tests.example.com' with connection received on either port 80 or 443 and where WSGI application was mounted at the URL of '/interpreter.wsgi'.

Single Or Multi Threaded

Apache supports differing Multiprocessing Modules (MPMs) having different attributes. One such difference is whether a specific Apache child process uses multiple threads for handling requests or whether a single thread is instead used.

Depending on how you configure a daemon process group when using daemon mode will also dictate whether single or multithreaded. By default, if number of threads is not explicitly specified for a daemon process group, it will be multithreaded.

Whether a WSGI application is executing within a multithreaded environment is important to know. If it is, then you need to ensure that your own code and any framework you are using is also thread safe.

A test WSGI script for validating whether WSGI application running in multithread configuration is as follows:

```
import sys

def application(environ, start_response):
    status = '200 OK'
    output = u'wsgi.multithread = %s' % repr(environ['wsgi.multithread'])

    response_headers = [('Content-type', 'text/plain'),
                        ('Content-Length', str(len(output)))]
    start_response(status, response_headers)

    return [output.encode('UTF-8')]
```

If multithreaded, this will yield:

```
wsgi.multithread = True
```

Multithreaded would usually be true on Windows, on UNIX if running in embedded mode and worker MPM is used by Apache, or if using daemon mode and number of threads not explicitly set, or number of threads explicitly set to value other than '1'.

Debugging Techniques

Be it when initially setting up mod_wsgi for the first time, or later during development or use of your WSGI application, you are bound to get some sort of unexpected Python error. By default all you are usually going to see as evidence of this is a HTTP 500 "Internal Server Error" being displayed in the browser window and little else.

The purpose of this document is to explain where to go look for more details on what caused the error, describe techniques one can use to have your WSGI application generate more useful debug information, as well as mechanisms that can be employed to interactively debug your application.

Note that although this document is intended to deal with techniques which can be used when using mod_wsgi, many of the techniques are also directly transferable or adaptable to other web hosting mechanisms for WSGI applications.

Apache Error Log Files

When using mod_wsgi, unless you or the web framework you are using takes specific action to catch exceptions and present the details in an alternate manner, the only place that details of uncaught exceptions will be recorded is in the Apache error log files. The Apache error log files are therefore your prime source of information when things go wrong.

Do note though that log messages generated by mod_wsgi are logged with various severity levels and which ones will be output to the Apache error log files will depend on how Apache has been configured. The standard configuration for Apache has the !LogLevel directive being set to 'warn'. With this setting any important error messages will be output, but informational messages generated by mod_wsgi which can assist in working out what it is doing are not. Thus, if new to mod_wsgi or trying to debug a problem, it is worthwhile setting the Apache configuration to use 'info' log level instead:

```
LogLevel info
```

If your Apache web server is only providing services for one host, it is likely that you will only have one error log file. If however the Apache web server is configured for multiple virtual hosts, then it is possible that there will be multiple error log files, one corresponding to the main server host and an additional error log file for each virtual host. Such a virtual host specific error log if one is being used, would have been configured through the placing of the Apache CustomLog directive within the context of the VirtualHost container.

Although your WSGI application may be hosted within a particular virtual host and that virtual host has its own error log file, some error and informational messages will still go to the main server host error log file. Thus you may still need to consult both error log files when using virtual hosts.

Messages of note that will end up in the main server host error log file include notifications in regard to initialisation of Python and the creation and destruction of Python sub interpreters, plus any errors which occur when doing this.

Messages of note that would end up in the virtual host error log file, if it exists, include details of uncaught Python exceptions which occur when the WSGI application script is being loaded, or when the WSGI application callable object is being executed.

Messages that are logged by a WSGI application via the 'wsgi.errors' object passed through to the application in the WSGI environment are also logged. These will go to the virtual host error log file if it exists, or the main error log file if the virtual host is not setup with its own error log file. Thus, if you want to add debugging messages to your WSGI application code, you can use 'wsgi.errors' in conjunction with the 'print' statement as shown below:

```
def application(environ, start_response):
    status = '200 OK'
    output = b'Hello World!'

    print >> environ['wsgi.errors'], "application debug #1"

    response_headers = [('Content-type', 'text/plain'),
                        ('Content-Length', str(len(output)))]
    start_response(status, response_headers)

    print >> environ['wsgi.errors'], "application debug #2"

    return [output]
```

If 'wsgi.errors' is not available to the code which needs to output log messages, then it should explicitly direct output from the 'print' statement to 'sys.stderr':

```
import sys

def function():
    print >> sys.stderr, "application debug #3"
        ...
```

If `sys.stderr` or `sys.stdout` is used directly then these messages will end up in the main server host error log file and not that for the virtual host unless the WSGI application is running in a daemon process specifically associated with a virtual host.

Do be aware though that writing to `sys.stdout` is by default restricted in versions of mod_wsgi prior to 3.0 and will result in an exception occurring of the form:

```
IOError: sys.stdout access restricted by mod_wsgi
```

This is because portable WSGI applications should not write to `sys.stdout` or use the 'print' statement without specifying an alternate file object besides `sys.stdout` as the target. This restriction can be disabled for the whole server using the WSGIRestrictStdout directive, or by mapping `sys.stdout` to `sys.stderr` at global scope within in the WSGI application script file:

```
import sys
sys.stdout = sys.stderr
```

In general, a WSGI application should always endeavour to only log messages via the 'wsgi.errors' object that is passed through to a WSGI application in the WSGI environment. This is because this is the only way of logging

messages for which there is some guarantee that they will end up in a log file that you might have access to if using a shared server.

An application shouldn't however cache 'wsgi.errors' and try to use it outside of the context of a request. If this is done an exception will be raised indicating that the request has expired and the error log object is now invalid.

That messages output via `sys.stderr` and `sys.stdout` end up in the Apache error logs at all is provided as a convenience but there is no requirement in the WSGI specification that they are valid means of a WSGI application logging messages.

Displaying Request Environment

When a WSGI application is invoked, the request headers are passed as CGI variables in the WSGI request environment. The dictionary used for this also holds information about the WSGI execution environment and mod_wsgi. This includes mod_wsgi specific variables indicating the name of the process and application groups within which the WSGI application is executing.

Knowing the values of the process and application group variables can be important when needing to validate that your Apache configuration is doing what you intended as far as ensuring your WSGI application is running in daemon mode or otherwise.

A simple way of validating such details or getting access to any of the other WSGI request environment variables is to substitute your existing WSGI application with one which echos back the details to your browser. Such a task can be achieved with the following test application. The application could be extended as necessary to display other information as well, with process ID, user ID and group ID being shown as examples:

```
import cStringIO
import os

def application(environ, start_response):
    headers = []
    headers.append(('Content-Type', 'text/plain'))
    write = start_response('200 OK', headers)

    input = environ['wsgi.input']
    output = cStringIO.StringIO()

    print >> output, "PID: %s" % os.getpid()
    print >> output, "UID: %s" % os.getuid()
    print >> output, "GID: %s" % os.getgid()
    print >> output

    keys = environ.keys()
    keys.sort()
    for key in keys:
        print >> output, '%s: %s' % (key, repr(environ[key]))
    print >> output

    output.write(input.read(int(environ.get('CONTENT_LENGTH', '0'))))

    return [output.getvalue()]
```

For the case of the process group as recorded by the 'mod_wsgi.process_group' variable in the WSGI request environment, if the value is an empty string then the WSGI application is running in embedded mode. For any other value it will be running in daemon mode with the process group named by the variables value.

Note that by default WSGI applications run in embedded mode, which means within the Apache server child processes which accept the original requests. Daemon mode processes would only be used through appropriate use of

the WSGIDaemonProcess and WSGIProcessGroup directives to delegate the WSGI application to a named daemon process group.

For the case of the application group as recorded by the 'mod_wsgi.application_group' variable in the WSGI request environment, if the value is an empty string then the WSGI application is running in the main Python interpreter. That is, the very first interpreter created when Python was initialised. For any other value it indicates it is running in the named Python sub interpreter.

Note that by default WSGI applications would always run in a sub interpreter rather than the main interpreter. The name of this sub interpreter would be automatically constructed from the name of the server or virtual host, the URL mount point of the WSGI application and the number of the listener port when it is other than ports 80 or 443.

To delegate a WSGI application to run in main Python interpreter, the WSGIApplicationGroup directive would need to have been used with the value '%{GLOBAL}'. Although the value is '%{GLOBAL}', this translates to the empty string seen for the value of 'mod_wsgi.application_group' within the WSGI request environment.

The WSGIApplicationGroup directive could also be used to designate a specific named sub interpreter rather than that selected automatically.

For newcomers this can all be a bit confusing, which is where the test application comes in as you can use it to validate where your WSGI application is running is where you intended it to run.

The set of WSGI request environment variables will also show the WSGI variables indicating whether process is multithreaded and whether the process group is multiprocess or not. For a more complete explanation of what that means see documentation of *Processes And Threading*.

Tracking Request and Response

Although one can use above test application to display the request environment, it is replacing your original WSGI application. Rather than replace your existing application you can use a WSGI middleware wrapper application which logs the details to the Apache error log instead:

```
# Original WSGI application.

def application(environ, start_response):
    ...

# Logging WSGI middleware.

import pprint

class LoggingMiddleware:

    def __init__(self, application):
        self.__application = application

    def __call__(self, environ, start_response):
        errors = environ['wsgi.errors']
        pprint.pprint(('REQUEST', environ), stream=errors)

        def _start_response(status, headers, *args):
            pprint.pprint(('RESPONSE', status, headers), stream=errors)
            return start_response(status, headers, *args)

        return self.__application(environ, _start_response)

application = LoggingMiddleware(application)
```

The output from the middleware would end up in the Apache error log for the virtual host, or if no virtual host specific error log file, in the main Apache error log file.

For more complicated problems it may also be necessary to track both the request and response content as well. A more complicated middleware which can log these as well as header information to the file system is as follows:

```
# Original WSGI application.

def application(environ, start_response):
    ...

# Logging WSGI middleware.

import threading
import pprint
import time
import os

class LoggingInstance:
    def __init__(self, start_response, oheaders, ocontent):
        self.__start_response = start_response
        self.__oheaders = oheaders
        self.__ocontent = ocontent

    def __call__(self, status, headers, *args):
        pprint.pprint((status, headers)+args), stream=self.__oheaders)
        self.__oheaders.close()

        self.__write = self.__start_response(status, headers, *args)
        return self.write

    def __iter__(self):
        return self

    def write(self, data):
        self.__ocontent.write(data)
        self.__ocontent.flush()
        return self.__write(data)

    def next(self):
        data = self.__iterable.next()
        self.__ocontent.write(data)
        self.__ocontent.flush()
        return data

    def close(self):
        if hasattr(self.__iterable, 'close'):
            self.__iterable.close()
        self.__ocontent.close()

    def link(self, iterable):
        self.__iterable = iter(iterable)

class LoggingMiddleware:

    def __init__(self, application, savedir):
        self.__application = application
        self.__savedir = savedir
        self.__lock = threading.Lock()
```

```
        self.__pid = os.getpid()
        self.__count = 0

    def __call__(self, environ, start_response):
        self.__lock.acquire()
        self.__count += 1
        count = self.__count
        self.__lock.release()

        key = "%s-%s-%s" % (time.time(), self.__pid, count)

        iheaders = os.path.join(self.__savedir, key + ".iheaders")
        iheaders_fp = file(iheaders, 'w')

        icontent = os.path.join(self.__savedir, key + ".icontent")
        icontent_fp = file(icontent, 'w+b')

        oheaders = os.path.join(self.__savedir, key + ".oheaders")
        oheaders_fp = file(oheaders, 'w')

        ocontent = os.path.join(self.__savedir, key + ".ocontent")
        ocontent_fp = file(ocontent, 'w+b')

        errors = environ['wsgi.errors']
        pprint.pprint(environ, stream=iheaders_fp)
        iheaders_fp.close()

        length = int(environ.get('CONTENT_LENGTH', '0'))
        input = environ['wsgi.input']
        while length != 0:
            data = input.read(min(4096, length))
            if data:
                icontent_fp.write(data)
                length -= len(data)
            else:
                length = 0
        icontent_fp.flush()
        icontent_fp.seek(0, os.SEEK_SET)
        environ['wsgi.input'] = icontent_fp

        iterable = LoggingInstance(start_response, oheaders_fp, ocontent_fp)
        iterable.link(self.__application(environ, iterable))
        return iterable

application = LoggingMiddleware(application, '/tmp/wsgi')
```

For this middleware, the second argument to the constructor should be a preexisting directory. For each request four files will be saved. These correspond to input headers, input content, response status and headers, and request content.

Poorly Performing Code

The WSGI specification allows any iterable object to be returned as the response, so long as the iterable yields string values. That this is the case means that one can too easily return an object which satisfies this requirement but has some sort of performance related issue.

The worst case of this is where instead of returning a list containing strings, a single string is returned. The problem with a string is that when it is iterated over, a single character of the string is yielded each time. In other words, a single

character is written back to the client on each loop, with a flush occurring in between to ensure that the character has actually been written and isn't just being buffered.

Although for small strings a performance impact may not be noticed, if returning large strings the affect on request throughput could be quite significant.

Another case which can cause problems is to return a file like object. For iteration over a file like object, typically what can occur is that a single line within the file is returned each time. If the file is a line oriented text file where each line is a of a reasonable length, this may be okay, but if the file is a binary file there may not actually be line breaks within the file.

For the case where file contains many short lines, throughput would be affected much like in the case where a string is returned. For the case where the file is just binary data, the result can be that the complete file may be read in on the first loop. If the file is large, this could cause a large transient spike in memory usage. Once that memory is allocated, it will then be retained by the process, albeit that it may be reused by the process at a later point.

Because of the performance impacts in terms of throughput and memory usage, both these cases should be avoided. For the case of returning a string, it should be returned with a single element list. For the case of a file like object, the 'wsgi.file_wrapper' extension should be used, or a wrapper which suitably breaks the response into chunks.

In order to identify where code may be inadvertently returning such iterable types, the following code can be used:

```
import types

import cStringIO
import socket
import StringIO

BAD_ITERABLES = [
  cStringIO.InputType,
  socket.SocketType,
  StringIO.StringIO,
  types.FileType,
  types.StringType,
]

class ValidatingMiddleware:

    def __init__(self, application):
        self.__application = application

    def __call__(self, environ, start_response):
        errors = environ['wsgi.errors']

        result = self.__application(environ, start_response)

        value = type(result)
        if value == types.InstanceType:
            value = result.__class__
        if value in BAD_ITERABLES:
            print >> errors, 'BAD ITERABLE RETURNED: ',
            print >> errors, 'URL=%s ' % environ['REQUEST_URI'],
            print >> errors, 'TYPE=%s' % value

        return result

def application(environ, start_response):
    ...

application = ValidatingMiddleware(application)
```

Error Catching Middleware

Because mod_wsgi only logs details of uncaught exceptions to the Apache error log and returns a generic HTTP 500 "Internal Server Error" response, if you want the details of any exception to be displayed in the error page and be visible from the browser, you will need to use a WSGI error catching middleware component.

One example of WSGI error catching middleware is the ErrorMiddleware class from Paste.

- http://www.pythonpaste.org

This class can be configured not only to catch exceptions and present the details to the browser in an error page, it can also be configured to send the details of any errors in email to a designated recipient, or log the details to an alternate log file.

Being able to have error details sent by email would be useful in a production environment or where your application is running on a web hosting environment and the Apache error logs would not necessarily be closely monitored on a day to day basis. Enabling of that particular feature though should possibly only be done when you have some confidence in the application else you might end up getting inundated with emails.

To use the error catching middleware from Paste you simply need to wrap your existing application with it such that it then becomes the top level application entry point:

```
def application(environ, start_response):
    status = '200 OK'
    output = b'Hello World!\n'

    response_headers = [('Content-type', 'text/plain'),
                        ('Content-Length', str(len(output)))]
    start_response(status, response_headers)

    return [output]

from paste.exceptions.errormiddleware import ErrorMiddleware
application = ErrorMiddleware(application, debug=True)
```

In addition to displaying information about the Python exception that has occurred and the stack traceback, this middleware component will also output information about the WSGI environment such that you can see what was being passed to the WSGI application. This can be useful if the cause of any problem was unexpected values passed in the headers of the HTTP request.

Note that error catching middleware is of absolutely no use for trying to capture and display in the browser any errors that occur at global scope within the WSGI application script when it is being imported. Details of any such errors occurring at this point will only be captured in the Apache error log files. As much as possible you should avoid performing complicated tasks when the WSGI application script file is being imported, instead you should only trigger such actions the first time a request is received. By doing this you will be able to capture errors in such initialisation code with the error catching middleware.

Also note that the debug mode whereby details are displayed in the browser should only be used during development and not in a production system. This is because details which are displayed may be of use to anyone who may wish to compromise your site.

Python Interactive Debugger

Python debuggers such as implemented by the 'pdb' module can sometimes be useful in debugging Python applications, especially where there is a need to single step through code and analyse application state at each point. Use of

such debuggers in web applications can be a bit more tricky than normal applications though and especially so with mod_wsgi.

The problem with mod_wsgi is that the Apache web server can create multiple child processes to respond to requests. Partly because of this, but also just to prevent problems in general, Apache closes off standard input at startup. Thus there is no actual way to interact with the Python debugger module if it were used.

To get around this requires having complete control of the Apache web server that you are using to host your WSGI application. In particular, it will be necessary to shutdown the web server and then startup the 'httpd' process explicitly in single process debug mode, avoiding the 'apachectl' management application altogether:

```
$ apachectl stop
$ httpd -X
```

If Apache is normally started as the 'root' user, this also will need to be run as the 'root' user otherwise the Apache web server will not have the required permissions to write to its log directories etc.

The result of starting the 'httpd' process in this way will be that the Apache web server will run everything in one process rather than using multiple processes. Further, it will not close off standard input thus allowing the Python debugger to be used.

Do note though that one cannot be using the ability of mod_wsgi to run your application in a daemon process when doing this. The WSGI application must be running within the main Apache process.

To trigger the Python debugger for any call within your code, the following customised wrapper for the 'Pdb' class should be used:

```
class Debugger:

    def __init__(self, object):
        self.__object = object

    def __call__(self, *args, **kwargs):
        import pdb, sys
        debugger = pdb.Pdb()
        debugger.use_rawinput = 0
        debugger.reset()
        sys.settrace(debugger.trace_dispatch)

        try:
            return self.__object(*args, **kwargs)
        finally:
            debugger.quitting = 1
            sys.settrace(None)
```

This might for example be used to wrap the actual WSGI application callable object:

```
def application(environ, start_response):
    status = '200 OK'
    output = b'Hello World!\n'

    response_headers = [('Content-type', 'text/plain'),
                        ('Content-Length', str(len(output)))]
    start_response(status, response_headers)

    return [output]

application = Debugger(application)
```

When a request is now received, the Python debugger will be triggered and you can interactively debug your application from the window you ran the 'httpd' process. For example:

```
> /usr/local/wsgi/scripts/hello.py(21)application()
-> status = '200 OK'

(Pdb) list
 16             finally:
 17                 debugger.quitting = 1
 18                 sys.settrace(None)
 19
 20     def application(environ, start_response):
 21  ->     status = '200 OK'
 22         output = b'Hello World!\n'
 23
 24         response_headers = [('Content-type', 'text/plain'),
 25                             ('Content-Length', str(len(output)))]
 26         start_response(status, response_headers)

(Pdb) print start_response
<built-in method start_response of mod_wsgi.Adapter object at 0x1160180>

cont
```

When wishing to allow the request to complete, issue the 'cont' command. If wishing to cause the request to abort, issue the 'quit' command. This will result in a 'BdbQuit' exception being raised and would result in a HTTP 500 "Internal Server Error" response being returned to the client. To kill off the whole 'httpd' process, after having issued 'cont' or 'quit' to exit the debugger, interrupt the process using 'CTRL-C'.

To see what commands the Python debugger accepts, issue the 'help' command and also consult the documentation for the 'pdb' module on the Python web site.

Note that the Python debugger expects to be able to write to `sys.stdout` to display information to the terminal. Thus if using using a Python web framework which replaces `sys.stdout` such as web.py, you will not be able to use the Python debugger.

Browser Based Debugger

In order to use the Python debugger modules you need to have direct access to the host and the Apache web server that is running your WSGI application. If your only access to the system is via your web browser this makes the use of the full Python debugger impractical.

An alternative to the Python debugger modules which is available is an extension of the WSGI error catching middleware previously described. This is the EvalException class from Paste. It embodies the error catching attributes of the ErrorMiddleware class, but also allows some measure of interactive debugging and introspection through the web browser.

As with any WSGI middleware component, to use the class entails creating a wrapper around the application you wish to debug:

```
def application(environ, start_response):
    status = '200 OK'
    output = b'Hello World!\n'

    response_headers = [('Content-type', 'text/plain'),
                        ('Content-Length', str(len(output)))]
    start_response(status, response_headers)
```

```
    return [output]

from paste.evalexception.middleware import EvalException
application = EvalException(application)
```

Like ErrorMiddleware when an unexpected exception occurs a web page is presented which shows the location of the error along with the contents of the WSGI application environment. Where EvalException is different however is that it is possible to inspect the local variables residing within each stack frame down to where the error occurred. Further, it is possible to enter Python code which can be evaluated within the context of the selected stack frame in order to access data or call functions or methods of objects.

In order for this to all work requires that subsequent requests back to the WSGI application always end up with the same process where the error originally occurred. With mod_wsgi this does however present a bit of a problem as Apache can create and use multiple child processes to handle requests.

Because of this requirement, if you want to be able to use this browser based interactive debugger, if running your application in embedded mode of mod_wsgi, you will need to configure Apache such that it only starts up one child process to handle requests and that it never creates any additional processes. The Apache configuration directives required to achieve this are as follows:

```
StartServers 1
ServerLimit 1
```

The directives must be placed at global scope within the main Apache configuration files and will affect the whole Apache web server.

If you are using the worker MPM on a UNIX system, restricting Apache to just a single process may not be an issue, at least during development. If however you are using the prefork MPM on a UNIX system, you may see issues if you are using an AJAX intensive page that relies on being able to execute parallel requests, as only one request at a time will be able to be handled by the Apache web server.

If using Apache 2.X on a UNIX system, a better approach is to use daemon mode of mod_wsgi and delegate your application to run in a single daemon process. This process may be single or multithreaded as per any threading requirements of your application.

Which ever configuration is used, if the browser based interactive debugger is used it should only be used on a development system and should never be deployed on a production system or in a web hosting environment. This is because the debugger will allow one to execute arbitrary Python code within the context of your application from a remote client.

Debugging Crashes With GDB

In cases where Apache itself crashes for no apparent reason, the above techniques are not always particularly useful. This is especially the case where the crash occurs in non Python code outside of your WSGI application.

The most common cause of Apache crashing, besides any still latent bugs that may exist in mod_wsgi, of which hopefully there aren't any, are shared library version mismatches. Another major cause of crashes is third party C extension modules for Python which are not compatible with being used in a Python sub interpreter which isn't the first interpreter created when Python is initialised, or modules which are not compatible with Python sub interpreters being destroyed and the module then being used in a new Python sub interpreter.

Examples of where shared library version mismatches are known to occur are between the version of the 'expat' library used by Apache and that embedded within the Python 'pyexpat' module. Another is between the version of the MySQL client libraries used by PHP and the Python MySQL module.

Both these can be a cause of crashes where the different components are compiled and linked against different versions of the shared library for the packages in question. It is vitally important that all packages making use of a shared library

were compiled against and use the same version of a shared library.

Another problematic package is Subversion. In this case there can be conflicts between the version of Subversion libraries used by mod_dav_svn and the Python Subversion bindings. Certain versions of the Python Subversion modules also cause problems because they appear to be incompatible with use in a Python sub interpreter which isn't the first interpreter created when Python is initialised.

In this latter issue, the sub interpreter problems can often be solved by forcing the WSGI application using the Python Subversion modules to run in the '%{GLOBAL}' application group. This solution often also resolves issues with SWIG generated bindings, especially where the `-thread` option was supplied to 'swig' when the bindings were generated.

Whatever the reason, in some cases the only way to determine why Apache or Python is crashing is to use a C code debugger such as 'gdb'. Now although it is possible to attach 'gdb' to a running process, the preferred method for using 'gdb' in conjunction with Apache is to run Apache in single process debug mode from within 'gdb'.

To do this it is necessary to first shutdown Apache. The 'gdb' debugger can then be started against the 'httpd' executable and then the process started up from inside of 'gdb':

```
$ /usr/local/apache/bin/apachectl stop
$ sudo gdb /usr/local/apache/bin/httpd
GNU gdb 6.1-20040303 (Apple version gdb-384) (Mon Mar 21 00:05:26 GMT 2005)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "powerpc-apple-darwin"...Reading symbols for shared
libraries ........ done

(gdb) run -X
Starting program: /usr/local/apache/bin/httpd -X
Reading symbols for shared libraries .+++ done
Reading symbols for shared libraries ..................... done
```

If Apache is normally started as the 'root' user, this also will need to be run as the 'root' user otherwise the Apache web server will not have the required permissions to write to its log directories etc.

If Apache was crashing on startup, you should immediately encounter the error, otherwise use your web browser to access the URL which is causing the crash to occur. You can then commence trying to debug why the crash is occurring.

Note that you should ensure that you have not assigned your WSGI application to run in a mod_wsgi daemon process using the WSGIDaemonProcess and WSGIProcessGroup directives. This is because the above procedure will only catch crashes which occur when the application is running in embedded mode. If it turns out that the application only crashes when run in mod_wsgi daemon mode, an alternate method of using 'gdb' will be required.

In this circumstance you should run Apache as normal, but ensure that you only create one mod_wsgi daemon process and have it use only a single thread:

```
WSGIDaemonProcess debug threads=1
WSGIProcessGroup debug
```

If not running the daemon process as a distinct user where you can tell which process it is, then you will also need to ensure that Apache !LogLevel directive has been set to 'info'. This is to ensure that information about daemon processes created by mod_wsgi are logged to the Apache error log. This is necessary, as you will need to consult the Apache error logs to determine the process ID of the daemon process that has been created for that daemon process group:

```
mod_wsgi (pid=666): Starting process 'debug' with threads=1.
```

Knowing the process ID, you should then run 'gdb', telling it to attach directly to the daemon process:

```
$ sudo gdb /usr/local/apache/bin/httpd 666
GNU gdb 6.1-20040303 (Apple version gdb-384) (Mon Mar 21 00:05:26 GMT 2005)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "powerpc-apple-darwin"...Reading symbols for shared
libraries ........ done

/Users/grahamd/666: No such file or directory.
Attaching to program: `/usr/local/apache/bin/httpd', process 666.
Reading symbols for shared libraries .+++..................... done
0x900c7060 in sigwait ()
(gdb) cont
Continuing.
```

Once 'gdb' has been started and attached to the process, then initiate the request with the URL that causes the application to crash.

Attaching to the running daemon process can also be useful where a single request or the whole process is appearing to hang. In this case one can force a stack trace to be output for all running threads to try and determine what code is getting stuck. The appropriate gdb command in this instance is 'thread apply all bt':

```
sudo gdb /usr/local/apache-2.2/bin/httpd 666
GNU gdb 6.3.50-20050815 (Apple version gdb-477) (Sun Apr 30 20:06:22 GMT 2006)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "powerpc-apple-darwin"...Reading symbols
for shared libraries ....... done

/Users/grahamd/666: No such file or directory.
Attaching to program: `/usr/local/apache/bin/httpd', process 666.
Reading symbols for shared libraries .+++..................... done
0x900c7060 in sigwait ()
(gdb) thread apply all bt

Thread 4 (process 666 thread 0xd03):
#0  0x9001f7ac in select ()
#1  0x004189b4 in apr_pollset_poll (pollset=0x1894650,
    timeout=-1146117585187099488, num=0xf0182d98, descriptors=0xf0182d9c)
    at poll/unix/select.c:363
#2  0x002a57f0 in wsgi_daemon_thread (thd=0x1889660, data=0x18895e8)
    at mod_wsgi.c:6980
#3  0x9002bc28 in _pthread_body ()

Thread 3 (process 666 thread 0xc03):
#0  0x9001f7ac in select ()
#1  0x0041d224 in apr_sleep (t=1000000) at time/unix/time.c:246
#2  0x002a2b10 in wsgi_deadlock_thread (thd=0x0, data=0x2aee68) at
    mod_wsgi.c:7119
```

```
#3  0x9002bc28 in _pthread_body ()

Thread 2 (process 666 thread 0xb03):
#0  0x9001f7ac in select ()
#1  0x0041d224 in apr_sleep (t=299970002) at time/unix/time.c:246
#2  0x002a2dec in wsgi_monitor_thread (thd=0x0, data=0x18890e8) at
    mod_wsgi.c:7197
#3  0x9002bc28 in _pthread_body ()

Thread 1 (process 666 thread 0x203):
#0  0x900c7060 in sigwait ()
#1  0x0041ba9c in apr_signal_thread (signal_handler=0x2a29a0
    <wsgi_check_signal>) at threadproc/unix/signals.c:383
#2  0x002a3728 in wsgi_start_process (p=0x1806418, daemon=0x18890e8)
    at mod_wsgi.c:7311
#3  0x002a6a4c in wsgi_hook_init (pconf=0x1806418, ptemp=0x0,
    plog=0xc8, s=0x18be8d4) at mod_wsgi.c:7716
#4  0x0000a5b0 in ap_run_post_config (pconf=0x1806418, plog=0x1844418,
    ptemp=0x180e418, s=0x180da78) at config.c:91
#5  0x000033d4 in main (argc=3, argv=0xbffffa8c) at main.c:706
```

It is suggested when trying to debug such issues that the daemon process be made to run with only a single thread. This will reduce how many stack traces one needs to analyse.

If you are running with multiple processes within the daemon process group and all requests are hanging, you will need to get a snapshot of what is happening in all processes in the daemon process group. Because doing this by hand will be tedious, it is better to automate it.

To automate capturing the stack traces, first create a file called 'gdb.cmds' which contains the following:

```
set pagination 0
thread apply all bt
detach
quit
```

This can then be used in conjunction with 'gdb' to avoid needing to enter the commands manually. For example:

```
sudo gdb /usr/local/apache-2.2/bin/httpd -x gdb.cmds -p 666
```

To be able to automate this further and apply it to all processes in a daemon process group, then first off ensure that daemon processes are named in 'ps' output by using the 'display-name' option to WSGIDaemonProcess directive.

For example, to apply default naming strategy as implemented by mod_wsgi, use:

```
WSGIDaemonProcess xxx display-name=%{GLOBAL}
```

In the output of a BSD derived 'ps' command, this will now show the process as being named '(wsgi:xxx)':

```
$ ps -cxo command,pid | grep wsgi
(wsgi:xxx)        666
```

Note that the name may be truncated as the resultant name can be no longer than what was the length of the original executable path for Apache. You may therefore like to name it explicitly:

```
WSGIDaemonProcess xxx display-name=(wsgi:xxx)
```

Having named the processes in the daemon process group, we can now parse the output of 'ps' to identify the process and apply the 'gdb' command script to each:

```
for pid in `ps -cxo command,pid | awk '{ if ($0 ~ /wsgi:xxx/ && $1 !~ /grep/) print
↪$NF }'`; do sudo gdb -x gdb.cmds -p $pid; done
```

The actual name given to the daemon process group using the 'display-name' option should be replaced in this command line. That is, change 'wsgi:xxx' appropriately.

If you are having problems with process in daemon process groups hanging, you might consider implementing a monitoring system which automatically detects somehow when the processes are no longer responding to requests and automatically trigger this dump of the stack traces before restarting the daemon process group or Apache.

Extracting Python Stack Traces

Using gdb to get stack traces as described above only gives you information about what is happening at the C code level. This will not tell where in the actual Python code execution was at. Your only clue is going to be where a call out was being made to some distinct C function in a C extension module for Python.

One can get stack traces for Python code by using:

```
def _stacktraces():
   code = []
   for threadId, stack in sys._current_frames().items():
       code.append("\n# ThreadID: %s" % threadId)
       for filename, lineno, name, line in traceback.extract_stack(stack):
           code.append('File: "%s", line %d, in %s' % (filename,
                   lineno, name))
           if line:
               code.append("  %s" % (line.strip()))

   for line in code:
       print >> sys.stderr, line
```

The caveat here obviously is that the process has to still be running. There is also the issue of how you trigger that function to dump stack traces for executing Python threads.

If the problem you have is that some request handler threads are stuck, either blocked, or stuck in an infinite loop, and you want to know what they are doing, then so long as there are still some handler threads left and the application is still responding to requests, then you could trigger it from a request handler triggered by making a request against a specific URL.

This though depends on you only running your application within a single process because as soon as you have multiple processes you have no guarantee that a request will go to the process you want to debug.

A better method therefore is to have a perpetually running background thread which monitors for a specific file in the file system. When that file is created or the modification time changes, then the background thread would dump the stack traces for the process.

Sample code which takes this approach is included below. This code could be placed temporarily at the end of your WSGI script file if you know you are going to need it because of a recurring problem:

```
import os
import sys
import time
import signal
import threading
import atexit
import Queue
import traceback
```

```
FILE = '/tmp/dump-stack-traces.txt'

_interval = 1.0

_running = False
_queue = Queue.Queue()
_lock = threading.Lock()

def _stacktraces():
    code = []
    for threadId, stack in sys._current_frames().items():
        code.append("\n# ProcessId: %s" % os.getpid())
        code.append("# ThreadID: %s" % threadId)
        for filename, lineno, name, line in traceback.extract_stack(stack):
            code.append('File: "%s", line %d, in %s' % (filename,
                    lineno, name))
            if line:
                code.append("  %s" % (line.strip()))

    for line in code:
        print >> sys.stderr, line

try:
    mtime = os.path.getmtime(FILE)
except:
    mtime = None

def _monitor():
    while 1:
        global mtime

        try:
            current = os.path.getmtime(FILE)
        except:
            current = None

        if current != mtime:
            mtime = current
            _stacktraces()

        # Go to sleep for specified interval.

        try:
            return _queue.get(timeout=_interval)
        except:
            pass

_thread = threading.Thread(target=_monitor)
_thread.setDaemon(True)

def _exiting():
    try:
        _queue.put(True)
    except:
        pass
    _thread.join()

atexit.register(_exiting)
```

```
def _start(interval=1.0):
    global _interval
    if interval < _interval:
        _interval = interval

    global _running
    _lock.acquire()
    if not _running:
        prefix = 'monitor (pid=%d):' % os.getpid()
        print >> sys.stderr, '%s Starting stack trace monitor.' % prefix
        _running = True
        _thread.start()
    _lock.release()

_start()
```

Once your WSGI script file has been loaded, then touching the file '/tmp/dump-stack-traces.txt' will cause stack traces for active Python threads to be output to the Apache error log.

Note that the sample code doesn't deal with possibility that with multiple processes for same application, that all processes may attempt to dump information at the same time. As such, you may get interleaving of output from multiple processes in Apache error logs at the same time.

What you may want to do is modify this code to dump out to some special directory, distinct files containing the trace where the names of the file include the process ID and a date/time. That way each will be separate.

An example of what one might expect to see from the above code is as follows:

```
# ProcessId: 666
# ThreadID: 4352905216
File: "/System/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/
↪threading.py", line 497, in __bootstrap
  self.__bootstrap_inner()
File: "/System/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/
↪threading.py", line 522, in __bootstrap_inner
  self.run()
File: "/System/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/
↪threading.py", line 477, in run
  self.__target(*self.__args, **self.__kwargs)
File: "/Library/WebServer/Sites/django-1/htdocs/project.wsgi", line 72, in _monitor
  _stacktraces()
File: "/Library/WebServer/Sites/django-1/htdocs/project.wsgi", line 47, in _
↪stacktraces
  for filename, lineno, name, line in traceback.extract_stack(stack):

# ThreadID: 4322832384
File: "/Library/WebServer/Sites/django-1/htdocs/project.wsgi", line 21, in application
  return _application(environ, start_response)
File: "/Library/WebServer/Sites/django-1/lib/python2.6/site-packages/django/core/
↪handlers/wsgi.py", line 245, in __call__
  response = middleware_method(request, response)
File: "/Library/WebServer/Sites/django-1/lib/python2.6/site-packages/django/contrib/
↪sessions/middleware.py", line 36, in process_response
  request.session.save()
File: "/Library/WebServer/Sites/django-1/lib/python2.6/site-packages/django/contrib/
↪sessions/backends/db.py", line 63, in save
  obj.save(force_insert=must_create, using=using)
File: "/Library/WebServer/Sites/django-1/lib/python2.6/site-packages/django/db/models/
↪base.py", line 434, in save
```

```
  self.save_base(using=using, force_insert=force_insert, force_update=force_update)
File: "/Library/WebServer/Sites/django-1/lib/python2.6/site-packages/django/db/models/
↪base.py", line 527, in save_base
  result = manager._insert(values, return_id=update_pk, using=using)
File: "/Library/WebServer/Sites/django-1/lib/python2.6/site-packages/django/db/models/
↪manager.py", line 195, in _insert
  return insert_query(self.model, values, **kwargs)
File: "/Library/WebServer/Sites/django-1/lib/python2.6/site-packages/django/db/models/
↪query.py", line 1479, in insert_query
  return query.get_compiler(using=using).execute_sql(return_id)
File: "/Library/WebServer/Sites/django-1/lib/python2.6/site-packages/django/db/models/
↪sql/compiler.py", line 783, in execute_sql
  cursor = super(SQLInsertCompiler, self).execute_sql(None)
File: "/Library/WebServer/Sites/django-1/lib/python2.6/site-packages/django/db/models/
↪sql/compiler.py", line 727, in execute_sql
  cursor.execute(sql, params)
File: "/Library/WebServer/Sites/django-1/lib/python2.6/site-packages/debug_toolbar/
↪panels/sql.py", line 95, in execute
  stacktrace = tidy_stacktrace(traceback.extract_stack())
File: "/Library/WebServer/Sites/django-1/lib/python2.6/site-packages/debug_toolbar/
↪panels/sql.py", line 40, in tidy_stacktrace
  s_path = os.path.realpath(s[0])
File: "/System/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/
↪posixpath.py", line 355, in realpath
  if islink(component):
File: "/System/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/
↪posixpath.py", line 132, in islink
  st = os.lstat(path)
```

Note that one of the displayed threads will be that for the thread which is dumping the stack traces. That stack trace can obviously be ignored.

One could extend the above recipe in more elaborate ways by using a WSGI middleware that capture details of each request from the WSGI environment and also dumping out from that the URL for the request being handled by any threads. This may assist in working out whether problems are related to a specific URL.

Processes And Threading

Apache can operate in a number of different modes dependent on the platform being used and the way in which it is configured. This ranges from multiple processes being used, with only one request being handled at a time within each process, to one or more processes being used, with concurrent requests being handled in distinct threads executing within those processes.

The combinations possible are further increased by mod_wsgi through its ability to create groups of daemon processes to which WSGI applications can be delegated. As with Apache itself, each process group can consist of one or more processes and optionally make use of multithreading. Unlike Apache, where some combinations are only possible based on how Apache was compiled, the mod_wsgi daemon processes can operate in any mode based only on runtime configuration settings.

This article provides background information on how Apache and mod_wsgi makes use of processes and threads to handle requests, and how Python sub interpreters are used to isolate WSGI applications. The implications of the various modes of operation on data sharing is also discussed.

WSGI Process/Thread Flags

Although Apache can make use of a combination of processes and/or threads to handle requests, this is not unique to the Apache web server and the WSGI specification acknowledges this fact. This acknowledgement is in the form of specific key/value pairs which must be supplied as part of the WSGI environment to a WSGI application. The purpose of these key/value pairs is to indicate whether the underlying web server does or does not make use of multiple processes and/or multiple threads to handle requests.

These key/value pairs are defined as follows in the WSGI specification.

wsgi.multithread This value should evaluate true if the application object may be simultaneously invoked by another thread in the same process, and should evaluate false otherwise.

wsgi.multiprocess This value should evaluate true if an equivalent application object may be simultaneously invoked by another process, and should evaluate false otherwise.

A WSGI application which is not written to take into consideration the different combinations of process and threading models may not be portable and potentially may not be robust when deployed to an alternate hosting platform or configuration.

Although you may not need an application or application component to work under all possible combinations for these values initially, it is highly recommended that any application component still be designed to work under any of the different operating modes. If for some reason this cannot be done due to the very nature of what functionality the component provides, the component should validate if it is being run within a compatible configuration and return a HTTP 500 internal server error response if it isn't.

An example of a component for which restrictions would apply is one providing an interactive browser based debugger session in response to an internal failure of a WSGI application. In this scenario, for the component to work correctly, subsequent HTTP requests must be processed by the same process. As such, the component can only be used with a web server that uses a single process. In other words, the value of 'wsgi.multiprocess' would have to evaluate to be false.

Multi-Processing Modules

The main factor which determines how Apache operates is which multi-processing module (MPM) is built into Apache at compile time. Although runtime configuration can customise the behaviour of the MPM, the choice of MPM will dictate whether or not multithreading is available.

On UNIX based systems, Apache defaults to being built with the 'prefork' MPM. If Apache 1.3 is being used this is actually the only choice, but for later versions of Apache, this can be overridden at build time by supplying an appropriate value in conjunction with the `--with-mpm` option when running the 'configure' script for Apache. The main alternative to the 'prefork' MPM which can be used on UNIX systems is the 'worker' MPM.

If you are unsure which MPM is built into Apache, it can be determined by running the Apache web server executable with the `-V` option. The output from running the web server executable with this option will be information about how it was configured when built:

```
Server version: Apache/2.2.1
Server built:   Mar  4 2007 20:48:15
Server's Module Magic Number: 20051115:1
Server loaded:  APR 1.2.6, APR-Util 1.2.6
Compiled using: APR 1.2.6, APR-Util 1.2.6
Architecture:   32-bit
Server MPM:     Worker
  threaded:     yes (fixed thread count)
    forked:     yes (variable process count)
Server compiled with....
 -D APACHE_MPM_DIR="server/mpm/worker"
```

```
-D APR_HAS_MMAP
-D APR_HAVE_IPV6 (IPv4-mapped addresses enabled)
-D APR_USE_SYSVSEM_SERIALIZE
-D APR_USE_PTHREAD_SERIALIZE
-D SINGLE_LISTEN_UNSERIALIZED_ACCEPT
-D APR_HAS_OTHER_CHILD
-D AP_HAVE_RELIABLE_PIPED_LOGS
-D DYNAMIC_MODULE_LIMIT=128
-D HTTPD_ROOT="/usr/local/apache-2.2"
-D SUEXEC_BIN="/usr/local/apache-2.2/bin/suexec"
-D DEFAULT_SCOREBOARD="logs/apache_runtime_status"
-D DEFAULT_ERRORLOG="logs/error_log"
-D AP_TYPES_CONFIG_FILE="conf/mime.types"
-D SERVER_CONFIG_FILE="conf/httpd.conf"
```

Which MPM is being used can be determined from the 'Server MPM' field.

On the Windows platform the only available MPM is 'winnt'.

The UNIX 'prefork' MPM

This MPM is the most commonly used. It was the only mode of operation available in Apache 1.3 and is still the default mode on UNIX systems in later versions of Apache. In this configuration, the main Apache process will at startup create multiple child processes. When a request is received by the parent process, it will be processed by which ever of the child processes is ready.

Each child process will only handle one request at a time. If another request arrives at the same time, it will be handled by the next available child process. When it is detected that the number of available processes is running out, additional child processes will be created as necessary. If a limit is specified as to the number of child processes which may be created and the limit is reached, plus there are sufficient requests arriving to fill up the listener socket queue, the client may instead receive an error resulting from not being able to establish a connection with the web server.

Where additional child processes have to be created due to a peak in the number of current requests arriving and where the number of requests has subsequently dropped off, the excess child processes may be shutdown and killed off. Child processes may also be shutdown and killed off after they have handled some set number of requests.

Although threads are not used to service individual requests, this does not preclude an application from creating separate threads to perform some specific task.

For the typical 'prefork' configuration where multiple processes are used, the WSGI environment key/value pairs indicating how processes and threads are being used will be as follows.

wsgi.multithread False

wsgi.multiprocess True

Because multiple processes are being used, a WSGI middleware component such as the interactive browser based debugger described would not be able to be used. If during development and testing of a WSGI application, use of such a debugger was required, the only option which would exist would be to limit the number of processes being used. This could be achieved using the Apache configuration:

```
StartServers 1
ServerLimit 1
```

With this configuration, only one process will be started, with no additional processes ever being created. The WSGI environment key/value pairs indicating how processes and threads are being used will for this configuration be as follows.

wsgi.multithread False

wsgi.multiprocess False

In effect, this configuration has the result of serialising all requests through a single process. This will allow an interactive browser based debugger to be used, but may prevent more complex WSGI applications which make use of AJAX techniques from working. This could occur where a web page initiates a sequence of AJAX requests and expects later requests to be able to complete while a response for an initial request is still pending. In other words, problems may occur where requests overlap, as subsequent requests will not be able to be executed until the initial request has completed.

The UNIX 'worker' MPM

The 'worker' MPM is similar to 'prefork' mode except that within each child process there will exist a number of worker threads. Instead of a request only being able to be processed by the next available idle child process and with the handling of the request being the only thing the child process is then doing, the request may be processed by a worker thread within a child process which already has other worker threads handling other requests at the same time.

It is possible that a WSGI application could be executed at the same time from multiple worker threads within the one child process. This means that multiple worker threads may want to access common shared data at the same time. As a consequence, such common shared data must be protected in a way that will allow access and modification in a thread safe manner. Normally this would necessitate the use of some form of synchronisation mechanism to ensure that only one thread at a time accesses and or modifies the common shared data.

If all worker threads within a child process were busy when a new request arrives the request would be processed by an idle worker thread in another child process. Apache may still create new child processes on demand if necessary. Apache may also still shutdown and kill off excess child processes, or child processes that have handled more than a set number of requests.

Overall, use of 'worker' MPM will result in less child processes needing to be created, but resource usage of individual child processes will be greater. On modern computer systems, the 'worker' MPM would in general be the prefered MPM to use and should if possible be used in preference to the 'prefork' MPM.

Although contention for the global interpreter lock (GIL) in Python can causes issues for pure Python programs, it is not generally as big an issue when using Python within Apache. This is because all the underlying infrastructure for accepting requests and mapping the URL to a WSGI application, as well as the handling of requests against static files are all performed by Apache in C code. While this code is being executed the thread will not be holding the Python GIL, thus allowing a greater level of overlapping execution where a system has multiple CPUs or CPUs with multiple cores.

This ability to make good use of more than processor, even when using multithreading, is further enchanced by the fact that Apache uses multiple processes for handling requests and not just a single process. Thus, even when there is some contention for the GIL within a specific process, it doesn't stop other processes from being able to run as the GIL is only local to a process and does not extend across processes.

For the typical 'worker' configuration where multiple processes and multiple threads are used, the WSGI environment key/value pairs indicating how processes and threads are being used will be as follows.

wsgi.multithread True

wsgi.multiprocess True

Similar to the 'prefork' MPM, the number of processes can be restricted to just one if required using the configuration:

```
StartServers 1
ServerLimit 1
```

With this configuration, only one process will be started, with no additional processes ever being created, but that one process would still make use of multiple threads.

The WSGI environment key/value pairs indicating how processes and threads are being used will for this configuration be as follows.

wsgi.multithread True

wsgi.multiprocess False

Because multiple threads are being used, there would be no problem with overlapping requests generated by an AJAX based web page.

The Windows 'winnt' MPM

On the Windows platform the 'winnt' MPM is the only option available. With this MPM, multiple worker threads within a child process are used to handle all requests. The 'winnt' MPM is different to the 'worker' mode however in that there is only one child process. At no time are additional child processes created, or that one child process shutdown and killed off, except where Apache as a whole is being stopped or restarted. Because there is only one child process, the maximum number of threads used is much greater.

The WSGI environment key/value pairs indicating how processes and threads are being used will for this configuration be as follows.

wsgi.multithread True

wsgi.multiprocess False

The mod_wsgi Daemon Processes

When using 'daemon' mode of mod_wsgi, each process group can be individually configured so as to run in a manner similar to either 'prefork', 'worker' or 'winnt' MPMs for Apache. This is achieved by controlling the number of processes and threads within each process using the 'processes' and 'threads' options of the WSGIDaemonProcess directive.

To emulate the same process/thread model as the 'winnt' MPM, that is, a single process with multiple threads, the following configuration would be used:

```
WSGIDaemonProcess example threads=25
```

The WSGI environment key/value pairs indicating how processes and threads are being used will for this configuration be as follows.

wsgi.multithread True

wsgi.multiprocess False

Note that by not specifying the 'processes' option only a single process is created within the process group. Although providing 'processes=1' as an option would also result in a single process being created, this has a slightly different meaning and so you should only do this if necessary.

The difference between not specifying the 'processes' option and defining 'processes=1' will be that WSGI environment attribute called 'wsgi.multiprocess' will be set to be True when the 'processes' option is defined, whereas not providing the option at all will result in the attribute being set to be False. This distinction is to allow for where some form of mapping mechanism might be used to distribute requests across multiple process groups and thus in effect it is still a multiprocess application.

In other words, if you use the configuration:

```
WSGIDaemonProcess example processes=1 threads=25
```

the WSGI environment key/value pairs indicating how processes and threads are being used will instead be:

wsgi.multithread True

wsgi.multiprocess True

If you need to ensure that 'wsgi.multiprocess' is False so that interactive debuggers do not complain about an incompatible configuration, simply do not specify the 'processes' option and allow the default behaviour of a single daemon process to apply.

To emulate the same process/thread model as the 'worker' MPM, that is, multiple processes with multiple threads, the following configuration would be used:

```
WSGIDaemonProcess example processes=2 threads=25
```

The WSGI environment key/value pairs indicating how processes and threads are being used will for this configuration be as follows.

wsgi.multithread True

wsgi.multiprocess True

To emulate the same process/thread model as the 'prefork' MPM, that is, multiple processes with only a single thread running in each, the following configuration would be used:

```
WSGIDaemonProcess example processes=5 threads=1
```

The WSGI environment key/value pairs indicating how processes and threads are being used will for this configuration be as follows.

wsgi.multithread False

wsgi.multiprocess True

Note that when using mod_wsgi daemon processes, the processes are only used to execute the Python based WSGI application. The processes are not in any way used to serve static files, or host applications implemented in other languages.

Unlike the normal Apache child processes when 'embedded' mode of mod_wsgi is used, the configuration as to the number of daemon processes within a process group is fixed. That is, when the server experiences additional load, no more daemon processes are created than what is defined. You should therefore always plan ahead and make sure the number of processes and threads defined is adequate to cope with the expected load.

Sharing Of Global Data

When the 'winnt' MPM is being used, or the 'prefork' or 'worker' MPM are forced to run with only a single process, all request handlers within a specific WSGI application will always be accessing the same global data. This global data will persist in memory until Apache is shutdown or restarted, or in the case of the 'prefork' or 'worker' MPM until the child process is recycled due to reaching a predefined request limit.

This ability to access the same global data and for that data to persist for the lifetime of the child process is not present when either of the 'prefork' or 'worker' MPM are used in multiprocess mode. In other words, where the WSGI environment key/value pair indicating how processes are used is set to:

wsgi.multiprocess True

This is because request handlers can execute within the context of distinct child processes, each with their own set of global data unique to that child process.

The consequences of this are that you cannot assume that separate invocations of a request handler will have access to the same global data if that data only resides within the memory of the child process. If some set of global data must be accessible by all invocations of a handler, that data will need to be stored in a way that it can be accessed from

multiple child processes. Such sharing could be achieved by storing the global data within an external database, the filesystem or in shared memory accessible by all child processes.

Since the global data will be accessible from multiple child processes at the same time, there must be adequate locking mechanisms in place to prevent distinct child processes from trying to modify the same data at the same time. The locking mechanisms need to also be able to deal with the case of multiple threads within one child process accessing the global data at the same time, as will be the case for the 'worker' and 'winnt' MPM.

Python Sub Interpreters

The default behaviour of mod_wsgi is to create a distinct Python sub interpreter for each WSGI application. Thus, where Apache is being used to host multiple WSGI applications a process will contain multiple sub interpreters. When Apache is run in a mode whereby there are multiple child processes, each child process will contain sub interpreters for each WSGI application.

When a sub interpreter is created for a WSGI application, it would then normally persist for the life of the process. The only exception to this would be where interpreter reloading is enabled, in which case the sub interpreter would be destroyed and recreated when the WSGI application script file has been changed.

For the sub interpreter created for each WSGI application, they will each have their own set of Python modules. In other words, a change to the global data within the context of one sub interpreter will not be seen from the sub interpreter corresponding to a different WSGI application. This will be the case whether or not the sub interpreters are in the same process.

This behaviour can be modified and multiple applications grouped together using the WSGIApplicationGroup directive. Specifically, the directive indicates that the marked WSGI applications should be run within the context of a common sub interpreter rather than being run in their own sub interpreters. By doing this, each WSGI application will then have access to the same global data. Do note though that this doesn't change the fact that global data will not be shared between processes.

The only other way of sharing data between sub interpreters within the one child process would be to use an external data store, or a third party C extension module for Python which allows communication or sharing of data between multiple interpreters within the same process.

Building A Portable Application

Taking into consideration the different process models used by Apache and the manner in which interpreters are used by mod_wsgi, to build a portable and robust application requires the following therefore be satisified.

1. Where shared data needs to be visible to all application instances, regardless of which child process they execute in, and changes made to the data by one application are immediately available to another, including any executing in another child process, an external data store such as a database or shared memory must be used. Global variables in normal Python modules cannot be used for this purpose.

2. Access to and modification of shared data in an external data store must be protected so as to prevent multiple threads in the same or different processes from interfering with each other. This would normally be achieved through a locking mechanism visible to all child processes.

3. An application must be re-entrant, or simply put, be able to be called concurrently by multiple threads at the same time. Data which needs to exist for the life of the request, would need to be stored as stack based data, thread local data, or cached in the WSGI application environment. Global variables within the actual application module cannot be used for this purpose.

4. Where global data in a module local to a child process is still used, for example as a cache, access to and modification of the global data must be protected by local thread locking mechanisms.

Reloading Source Code

This document contains information about mechanisms available in mod_wsgi for automatic reloading of source code when an application is changed and any issues related to those mechanisms.

Embedded Mode Vs Daemon Mode

What is achievable in the way of automatic source code reloading depends on which mode your WSGI application is running.

If your WSGI application is running in embedded mode then what happens when you make code changes is largely dictated by how Apache works, as it controls the processes handling requests. In general, if using embedded mode you will have no choice but to manually restart Apache in order for code changes to be used.

If using daemon mode, because mod_wsgi manages directly the processes handling requests and in which your WSGI application runs, there is more avenue for performing automatic source code reloading.

As a consequence, it is important to understand what mode your WSGI application is running in.

If you are running on Windows, are using Apache 1.3, or have not used WSGIDaemonProcess/WSGIProcessGroup directives to delegate your WSGI application to a mod_wsgi daemon mode process, then you will be using embedded mode.

If you are not sure whether you are using embedded mode or daemon mode, then substitute your WSGI application entry point with:

```
def application(environ, start_response):
    status = '200 OK'

    if not environ['mod_wsgi.process_group']:
      output = u'EMBEDDED MODE'
    else:
      output = u'DAEMON MODE'

    response_headers = [('Content-Type', 'text/plain'),
                        ('Content-Length', str(len(output)))]

    start_response(status, response_headers)

    return [output.encode('UTF-8')]
```

If your WSGI application is running in embedded mode, this will output to the browser 'EMBEDDED MODE'. If your WSGI application is running in daemon mode, this will output to the browser 'DAEMON MODE'.

Reloading In Embedded Mode

However you have configured Apache to mount your WSGI application, you will have a script file which contains the entry point for the WSGI application. This script file is not treated exactly like a normal Python module and need not even use a '.py' extension. It is even preferred that a '.py' extension not be used for reasons described below.

For embedded mode, one of the properties of the script file is that by default it will be reloaded whenever the file is changed. The primary intent with the file being reloaded is to provide a second chance at getting any configuration in it and the mapping to the application correct. If the script weren't reloaded in this way, you would need to restart Apache even for a trivial change to the script file.

Do note though that this script reloading mechanism is not intended as a general purpose code reloading mechanism. Only the script file itself is reloaded, no other Python modules are reloaded. This means that if modifying normal

Python code files which are used by your WSGI application, you will need to trigger a restart of Apache. For example, if you are using Django in embedded mode and needed to change your 'settings.py' file, you would still need to restart Apache.

That only the script file and not the whole process is reloaded also has a number of implications and imposes certain restrictions on what code in the script file can do or how it should be implemented.

The first issue is that when the script file is imported, if the code makes modifications to `sys.path` or other global data structures and the changes are additive, checks should first be made to ensure that the change has not already been made, else duplicate data will be added every time the script file is reloaded.

This means that when updating `sys.path`, instead of using:

```
import sys
sys.path.append('/usr/local/wsgi/modules')
```

the more correct way would be to use:

```
import sys
path = '/usr/local/wsgi/modules'
if path not in sys.path:
    sys.path.append(path)
```

This will ensure that the path doesn't get added multiple times.

Even where the script file is named so as to have a '.py' extension, that the script file is not treated like a normal module means that you should never try to import the file from another code file using the 'import' statement or any other import mechanism. The easiest way to avoid this is not use the '.py' extension on script files or never place script files in a directory which is located on the standard module search path, nor add the directory containing the script into `sys.path` explicitly.

If an attempt is made to import the script file as a module the result will be that it will be loaded a second time as an independent module. This is because script files are loaded under a module name which is keyed to the full absolute path for the script file and not just the basename of the file. Importing the script file directly and accessing it will therefore not result in the same data being accessed as exists in the script file when loaded.

Because the script file is not treated like a normal Python module also has implications when it comes to using the "pickle" module in conjunction with objects contained within the script file.

In practice what this means is that neither function objects, class objects or instances of classes which are defined in the script file should be stored using the "pickle" module.

The technical reasons for the limitations on the use of the "pickle" module in conjunction with objects defined in the script file are further discussed in the document *Issues With Pickle Module*.

The act of reloading script files also means that any data previously held by the module corresponding to the script file will be deleted. If such data constituted handles to database connections, and the connections are not able to clean up themselves when deleted, it may result in resource leakage.

One should therefore be cautious of what data is kept in a script file. Preferably the script file should only act as a bridge to code and data residing in a normal Python module imported from an entirely different directory.

Restarting Apache Processes

As explained above, the only facility that mod_wsgi provides for reloading source code files in embedded mode, is the reloading of just the script file providing the entry point for your WSGI application.

If you don't have a choice but to use embedded mode and still desire some measure of automatic source code reloading, one option available which works for both Windows and UNIX systems is to force Apache to recycle the Apache server child process that handles the request automatically after the request has completed.

To enable this, you need to modify the value of the MaxRequestsPerChild directive in the Apache configuration. Normally this would be set to a value of '0', indicating that the process should never be restarted as a result of the number of requests processed. To have it restart a process after every request, set it to the value '1' instead:

```
MaxRequestsPerChild 1
```

Do note however that this will cause the process to be restarted after any request. That is, the process will even be restarted if the request was for a static file or a PHP application and wasn't even handled by your WSGI application. The restart will also occur even if you have made no changes to your code.

Because a restart happens regardless of the request type, using this method is not recommended.

Because of how the Apache server child processes are monitored and restarts handled, it is technically possible that this method will yield performance which is worse than CGI scripts. For that reason you may even be better off using a CGI/WSGI bridge to host your WSGI application. At least that way the handling of other types of requests, such as for static files and PHP applications will not be affected.

Reloading In Daemon Mode

If using mod_wsgi daemon mode, what happens when the script file is changed is different to what happens in embedded mode. In daemon mode, if the script file changed, rather than just the script file being reloaded, the daemon process which contains the application will be shutdown and restarted automatically.

Detection of the change in the script file will occur at the time of the first request to arrive after the change has been made. The way that the restart is performed does not affect the handling of the request, with it still being processed once the daemon process has been restarted.

In the case of there being multiple daemon processes in the process group, then a cascade effect will occur, with successive processes being restarted until the request is again routed to one of the newly restarted processes.

In this way, restarting of a WSGI application when a change has been made to the code is a simple matter of touching the script file if daemon mode is being used. Any daemon processes will then automatically restart without the need to restart the whole of Apache.

So, if you are using Django in daemon mode and needed to change your 'settings.py' file, once you have made the required change, also touch the script file containing the WSGI application entry point. Having done that, on the next request the process will be restarted and your Django application reloaded.

Restarting Daemon Processes

If you are using daemon mode of mod_wsgi, restarting of processes can to a degree also be controlled by a user, or by the WSGI application itself, without restarting the whole of Apache.

To force a daemon process to be restarted, if you are using a single daemon process with many threads for the application, then you can embed a page in your application (password protected hopefully), that sends an appropriate signal to itself.

This should only be done for daemon processes and not within the Apache child processes, as sending such a signal within a child process may interfere with the operation of Apache. That the code is executing within a daemon process can be determined by checking the 'mod_wsgi.process_group' variable in the WSGI environment passed to the application. The value will be non empty if a daemon process:

```
if environ['mod_wsgi.process_group'] != '':
    import signal, os
    os.kill(os.getpid(), signal.SIGINT)
```

This will cause the daemon process your application is in to shutdown. The Apache process supervisor will then automatically restart your process ready for subsequent requests. On the restart it will pick up your new code. This way you can control a reload from your application through some special web page specifically for that purpose.

You can also send this signal from an external application, but a problem there may be identifying which process to send the signal to. If you are running the daemon process(es) as a distinct user/group to Apache and each application is running as a different user then you could just look for the Apache (httpd) processes owned by the user the application is running as, as opposed to the Apache user, and send them all signals.

If the daemon process is running as the same user as Apache or there are distinct applications running in different daemon processes but as the same user, knowing which daemon processes to send the signal may be harder to determine.

Either way, to make it easier to identify which processes belong to a daemon process group, you can use the 'display-name' option to the WSGIDaemonProcess to name the process. On many platforms, when this option is used, that name will then appear in the output from the 'ps' command and not the name of the actual Apache server binary.

Monitoring For Code Changes

The use of signals to restart a daemon process could also be employed in a mechanism which automatically detects changes to any Python modules or dependent files. This could be achieved by creating a thread at startup which periodically looks to see if file timestamps have changed and trigger a restart if they have.

Example code for such an automatic restart mechanism which is compatible with how mod_wsgi works is shown below:

```
from __future__ import print_function

import os
import sys
import time
import signal
import threading
import atexit

try:
    import Queue as queue
except ImportError:
    import queue

_interval = 1.0
_times = {}
_files = []

_running = False
_queue = queue.Queue()
_lock = threading.Lock()

def _restart(path):
    _queue.put(True)
    prefix = 'monitor (pid=%d):' % os.getpid()
    print('%s Change detected to \'%s\'.' % (prefix, path), file=sys.stderr)
    print('%s Triggering process restart.' % prefix, file=sys.stderr)
    os.kill(os.getpid(), signal.SIGINT)

def _modified(path):
    try:
        # If path doesn't denote a file and were previously
        # tracking it, then it has been removed or the file type
```

```
        # has changed so force a restart. If not previously
        # tracking the file then we can ignore it as probably
        # pseudo reference such as when file extracted from a
        # collection of modules contained in a zip file.

        if not os.path.isfile(path):
            return path in _times

        # Check for when file last modified.

        mtime = os.stat(path).st_mtime
        if path not in _times:
            _times[path] = mtime

        # Force restart when modification time has changed, even
        # if time now older, as that could indicate older file
        # has been restored.

        if mtime != _times[path]:
            return True
    except:
        # If any exception occured, likely that file has been
        # been removed just before stat(), so force a restart.

        return True

    return False

def _monitor():
    while 1:
        # Check modification times on all files in sys.modules.

        for module in sys.modules.values():
            if not hasattr(module, '__file__'):
                continue
            path = getattr(module, '__file__')
            if not path:
                continue
            if os.path.splitext(path)[1] in ['.pyc', '.pyo', '.pyd']:
                path = path[:-1]
            if _modified(path):
                return _restart(path)

        # Check modification times on files which have
        # specifically been registered for monitoring.

        for path in _files:
            if _modified(path):
                return _restart(path)

        # Go to sleep for specified interval.

        try:
            return _queue.get(timeout=_interval)
        except:
            pass

_thread = threading.Thread(target=_monitor)
```

```
_thread.setDaemon(True)

def _exiting():
    try:
        _queue.put(True)
    except:
        pass
    _thread.join()

atexit.register(_exiting)

def track(path):
    if not path in _files:
        _files.append(path)

def start(interval=1.0):
    global _interval
    if interval < _interval:
        _interval = interval

    global _running
    _lock.acquire()
    if not _running:
        prefix = 'monitor (pid=%d):' % os.getpid()
        print('%s Starting change monitor.' % prefix, file=sys.stderr)
        _running = True
        _thread.start()
    _lock.release()
```

This would be used by importing into the script file the Python module containing the above code, starting the monitoring system and adding any additional non Python files which should be tracked:

```
import os

import monitor
monitor.start(interval=1.0)
monitor.track(os.path.join(os.path.dirname(__file__), 'site.cf'))

def application(environ, start_response):
    ...
```

Where needing to add many non Python files in a directory hierarchy, such as template files which would otherwise be cached within the running process, the `os.path.walk()` function could be used to traverse all files and add required files based on extension or other criteria using the 'track()' function.

This mechanism would generally work adequately where a single daemon process is used within a process group. You would need to be careful however when multiple daemon processes are used. This is because it may not be possible to synchronise the checks exactly across all of the daemon processes. As a result you may end up with the daemon processes running a mixture of old and new code until they all synchronise with the new code base. This problem can be minimised by defining a short interval time between scans, however that will increase the overhead of the checks.

Using such an approach may in some cases be useful if using mod_wsgi as a development platform. It certainly would not be recommended you use this mechanism for a production system.

The reasons for not using it on a production system is due to the additional overhead and chance that daemon processes are restarted when you are not expecting them to be. For example, in a production environment where requests are coming in all the time, you do not want a restart triggered when you are part way through making a set of changes which cover multiple files as likely then that an inconsistent set of code will be loaded and the application will fail.

Note that you should also not use this mechanism on a system where you have configured mod_wsgi to preload your WSGI application as soon as the daemon process has started. If you do that, then the monitor thread will be recreated immediately and so for every single code change on a preloaded file you make, the daemon process will be restarted, even if there is no intervening request.

If preloading was really required, the example code would need to be modified so as to not use signals to restart the daemon process, but reset to zero the variable saved away in the WSGI script file that records the modification time of the script file. This will have the affect of delaying the restart until the next request has arrived. Because that variable holding the modification time is an internal implementation detail of mod_wsgi and not strictly part of its published API or behaviour, you should only use that approach if it is warranted.

Restarting Windows Apache

On the Windows platform there is no daemon mode only embedded mode. The MPM used on Apache is the 'winnt' MPM. This MPM is like the worker MPM on UNIX systems except that there is only one process.

Being embedded mode, modifying the WSGI script file only results in the WSGI script file itself being reloaded, the process as a whole is not reloaded. Thus there is no way normally through modifying the WSGI script file or any other Python code file used by the application, of having the whole application reloaded automatically.

The recipe in the previous section can be used with daemon mode on UNIX systems to implement an automated scheme for restarting the daemon processes when any code change is made, but because Windows lacks the 'fork()' system call daemon mode isn't supported in the first place.

Thus, the only way one can have code changes picked up on Windows is to restart Apache as a whole. Although a full restart is required, Apache on Windows only uses a single child server process and so the impact isn't as significant as on UNIX platforms, where many processes may need to be shutdown and restarted.

With that in mind, it is actually possible to modify the prior recipe for restarting a daemon process to restart Apache itself. To achieve this slight of hand, it is necessary to use the Python 'ctypes' module to get access to a special internal Apache function which is available in the Windows version of Apache called 'ap_signal_parent()'.

The required change to get this to work is to replace the restart function in the previous code with the following:

```
def _restart(path):
    _queue.put(True)
    prefix = 'monitor (pid=%d):' % os.getpid()
    print('%s Change detected to \'%s\'.' % (prefix, path), file=sys.stderr)
    print('%s Triggering Apache restart.' % prefix, file=sys.stderr)
    import ctypes
    ctypes.windll.libhttpd.ap_signal_parent(1)
```

Other than that, the prior code would be used exactly as before. Now when any change is made to Python code used by the application or any other monitored files, Apache will be restarted automatically for you.

As before, probably recommended that this only be used during development and not on a production system.

Virtual Environments

This document contains information about how to use Python virtual environments with mod_wsgi. You can use a Python virtual environment created using virtualenv and virtualenvwrapper, or if using Python 3, the `pyvenv` or `python -m venv` commands.

The purpose of a Python virtual environments is to allow one to create multiple distinct Python environments for the same version of Python, but with different sets of Python modules and packages installed. It is recommended that you always use Python virtual environments and not install additional Python packages direct into your Python installation.

A Python virtual environment is also required where it is necessary to run multiple WSGI applications which have conflicting requirements as to what version of a Python module or package needs to be installed. They can also be used when distinct mod_wsgi daemon process groups are used to host WSGI applications for different users and each user needs to be able to separately install their own Python modules and packages.

How you configure mod_wsgi or setup your WSGI application script file for a Python virtual environment will depend on your specific requirements. The more common scenarios are explained below.

Location of the Virtual Environment

Whichever method you use to create a Python virtual environment, before you use it with mod_wsgi, you should validate what the location of the Python virtual environment is. If using virtualenvwrapper this may be a non obvious directory hidden away under your home directory.

The way to determine the location of the Python virtual environment is to activate the Python virtual environment from an interactive shell so it is being used, and then run the command:

```
python -c 'import sys; print(sys.prefix)'
```

This will output the directory path you will use when setting up mod_wsgi to use the Python virtual environment. For the purposes of the examples below, it is assumed the location of any Python virtual environments are under the `/usr/local/venvs` directory. A specific Python virtual environment may thus return for `sys.prefix`:

```
/usr/local/venvs/example
```

Note that this should be the root directory of the Python virtual environment, which in turn contains the `bin` and `lib` directories for the Python virtual environment. It is a common mistake when setting up a Python virtual environment with mod_wsgi to use the full path to the `python` executable instead of the root directory. That will not work, so do not use the path for the `python` executable as the location of the Python virtual environment, it has to be the root directory.

Do be aware that the user that Apache runs your code as will need to be able to access the Python virtual environment. On some Linux distributions, the home directory of a user account is not accessible to other users. Rather than change the permissions on your home directory, it might be better to consider locating your WSGI application code and any Python virtual environment outside of your home directory.

Virtual Environment and Python Version

When using a Python virtual environment with mod_wsgi, it is very important that it has been created using the same Python installation that mod_wsgi was originally compiled for. It is not possible to use a Python virtual environment to force mod_wsgi to use a different Python version, or even a different Python installation.

You cannot for example force mod_wsgi to use a Python virtual environment created using Python 3.5 when mod_wsgi was originally compiled for Python 2.7. This is because the Python library for the Python installation it was originally compiled against is linked directly into the mod_wsgi module. In other words, Python is embedded within mod_wsgi. When mod_wsgi is used it does not run the command line `python` program to run the interpreter and thus why you can't force it to use a different Python installation.

The problem in trying to force mod_wsgi to use a different Python installation than what it was compiled for, even where it is the same Python version, is that the Python installation may itself not have been compiled with the same options. This is especially a problem when it comes to issues around how Python stores Unicode characters in memory.

The end result is that if you want to use a different Python installation or version than what mod_wsgi was originally compiled for, you would need to re-install mod_wsgi such that it is compiled for the Python installation or version you do want to use. Do not try and use a Python virtual environment from one Python installation or version with mod_wsgi, when mod_wsgi was compiled for a different one.

Daemon Mode (Single Application)

The preferred way of setting up mod_wsgi is to run each WSGI application in its own daemon process group. This is called daemon mode. A typical configuration for running a WSGI application in daemon mode would be:

```
WSGIDaemonProcess myapp

WSGIProcessGroup myapp
WSGIApplicationGroup %{GLOBAL}

WSGIScriptAlias / /some/path/project/myapp.wsgi

<Directory /some/path/project>
    Require all granted
</Directory>
```

The `WSGIDaemonProcess` directive defines the daemon process group. The `WSGIProcessGroup` directive indicates that the WSGI application should be run within the defined daemon process group.

As only the single application is being run within the daemon process group, the `WSGIApplicationGroup` directive is also being used. When this is used with the `%{GLOBAL}` value, it forces the WSGI application to run in the main Python interpreter context of each process. This is preferred in this scenario as some third party packages for Python which include C extensions will not run in the Python sub interpreter contexts which mod_wsgi would use by default. By using the main Python interpreter context you eliminate the possibility of such third party packages for Python causing problems.

To modify the configuration for this scenario to use a Python virtual environment, all you need to do is add the `python-home` option to the `WSGIDaemonProcess` directive resulting in:

```
WSGIDaemonProcess myapp python-home=/usr/local/venvs/myapp
```

All the additonal Python packages and modules would then be installed into that Python virtual environment.

Daemon Mode (Multiple Applications)

If instead of running each WSGI application in a separate daemon process group as is the recommended practice, you are running multiple WSGI applications in one daemon process group, a different approach to using Python virtual environments is required.

For this scenario there are various ways the configuration could be set up. If mounting each WSGI application explicitly you might be using:

```
WSGIDaemonProcess myapps

WSGIProcessGroup myapps

WSGIScriptAlias /myapp3 /some/path/project/myapp3.wsgi
WSGIScriptAlias /myapp2 /some/path/project/myapp2.wsgi

WSGIScriptAlias / /some/path/project/myapp1.wsgi

<Directory /some/path/project>
    Require all granted
</Directory>
```

If instead the directory containing the WSGI application script files is being mounted, you might be using:

```
WSGIDaemonProcess myapps

WSGIProcessGroup myapps

WSGIScriptAlias / /some/path/project/

<Directory /some/path/project>
    Require all granted
</Directory>
```

The use of the `WSGIDaemonProcess` and `WSGIProcessGroup` is the same as before, however the `WSGIApplicationGroup` directive is not being used.

When the `WSGIApplicationGroup` directive isn't being used to override which Python interpreter context is being used, each WSGI application will be run in its own Python sub interpreter context of the processes. This is necessary as often WSGI application frameworks (Django being a prime example), do not support running more than one instance of a WSGI application using the framework, in the same Python interpreter context at the same time.

In this scenario of running multiple WSGI applications in the same daemon process group, more than one change is possibly required. The changes required depend on whether or not all WSGI applications should share the same Python virtual environment.

If all of the WSGI applications should share the same Python virtual environment, then the same change as was performed above for the single application case would be made. That is, add the `python-home` option to the `WSGIDaemonProcess` directive:

```
WSGIDaemonProcess myapp python-home=/usr/local/venvs/myapps
```

All the additonal Python packages and modules that any of the WSGI applications required would then be installed into that Python virtual environment. Because it is a shared environment, they must all use the same version of any specific Python package or module.

If instead of all WSGI applications using the same Python virtual environment each needed their own, then a change will instead need to be made in each of the WSGI script files for the applications.

How this is done will depend on how the Python virtual environment is created.

If the Python virtual environment is created using virtualenv or virtualenvwrapper, the WSGI script for each application should be modified to include code of the following form:

```
python_home = '/usr/local/envs/myapp1'

activate_this = python_home + '/bin/activate_this.py'
execfile(activate_this, dict(__file__=activate_this))
```

Because each WSGI application is to use a separate Python virtual environment, the value of the `python_home` variable would be set differently for each WSGI script file, with it referring to the root directory of the respective Python virtual environments.

This code should be placed in the WSGI script file before any other module imports in the WSGI script file, with the exception of `from __future__` imports used to enable Python feature flags.

Important to note is that when the Python virtual environment is activated from within the WSGI script, what happens is a bit different to when the `python-home` option to `WSGIDaemonProcess` is used.

When activating the Python virtual environment from within the WSGI script file, only the `site-packages` directory from the Python virtual environment is being used. This directory will be added to the Python module search path, along with any additional directories related to the `site-packages` directory registered using `.pth` files present in the `site-packages` directory. This will be placed at the start of the existing `sys.path`.

The consequence of this is that the Python virtual environment isn't completely overriding the original Python installation the Python virtual environment was created from. This means that if the main Python installation had additional Python packages installed they will also potentially be visible to the WSGI application.

That this occurs could cause confusion as you might for example think you had all the packages you require listed in your `requirements.txt` file for `pip`, but didn't and so a package may not have been installed. If that package was installed in the main Python installation, it would be picked up from there, but it might be the wrong version and have dependencies on versions of other packages for which you have different versions installed in your Python virtual environment and which are found instead of those in the main Python installation.

To avoid such problems, when activating the Python virtual environment from within the WSGI script file, it is necessary to still set the `python-home` option of the `WSGIDaemonProcess` directive, but set it to an empty Python virtual environment which has had no additional packages installed:

```
WSGIDaemonProcess myapp python-home=/usr/local/venvs/empty
```

By doing this, the main Python installation will not be consulted and instead it will fallback to the empty Python virtual environment. This Python virtual environment should remain empty and you should not install additional Python packages or modules into it, or you will cause the same sort of conflicts that can arise with the main Python installation when it was being used.

When needing to activate the Python virtual environment from within the WSGI script file as described, it is preferred that you be using the either virtualenv or virtualenvwrapper to create the Python virtual environment. This is because they both provide the `activate_this.py` script file which does all the work of setting up `sys.path`. When you use either `pyvenv` or `python -m venv` with Python 3, no such activation script is provided.

So use virtualenv or virtualenvwrapper if you can. If you cannot for some reason and are stuck with `pyvenv` or `python -m venv`, you can instead use the following code in the WSGI script file:

```
python_home = '/usr/local/envs/myapp1'

import sys
import site

# Calculate path to site-packages directory.

python_version = '.'.join(map(str, sys.version_info[:2]))
site_packages = python_home + '/lib/python%s/site-packages' % python_version

# Add the site-packages directory.

site.addsitedir(site_packages)
```

As before this code should be placed in the WSGI script file before any other module imports in the WSGI script file, with the exception of `from __future__` imports used to enable Python feature flags.

When using this method, do be aware that the additions to the Python module search path are made at the end of `sys.path`. For that reason, you must set the `python-home` option to `WSGIDaemonProcess` to the location of an empty Python virtual environment. If you do not do this, any additional Python package installed in the main Python installation will hide those in the Python virtual environment for the application.

There is extra code you could add which would reorder `sys.path` to make it work in an equivalent way to the `activate_this.py` script provided when you use virtualenv or virtualenvwrapper but it is messy and more trouble than it is worth:

```
python_home = '/usr/local/envs/myapp1'

import sys
import site
```

```
# Calculate path to site-packages directory.

python_version = '.'.join(map(str, sys.version_info[:2]))
site_packages = python_home + '/lib/python%s/site-packages' % python_version
site.addsitedir(site_packages)

# Remember original sys.path.

prev_sys_path = list(sys.path)

# Add the site-packages directory.

site.addsitedir(site_packages)

# Reorder sys.path so new directories at the front.

new_sys_path = []

for item in list(sys.path):
    if item not in prev_sys_path:
        new_sys_path.append(item)
        sys.path.remove(item)

sys.path[:0] = new_sys_path
```

It is better to avoid needing to manually activate the Python virtual environment from inside of a WSGI script by using a separate daemon process group per WSGI application. At the minimum, at least avoid `pyvenv` and `python -m venv`.

Embedded Mode (Single Application)

The situation for running a single WSGI application in embedded mode is not much different to running a single WSGI application in daemon mode. In the case of embedded mode, there is though no `WSGIDaemonProcess` directive.

The typical configuration when running a single WSGI application in embedded module might be:

```
WSGIScriptAlias / /some/path/project/myapp.wsgi

WSGIApplicationGroup %{GLOBAL}

<Directory /some/path/project>
    Require all granted
</Directory>
```

The `WSGIDaemonProcess` and `WSGIProcessGroup` directives are gone, but the `WSGIApplicationGroup` directive is still used to force the WSGI application to run in the main Python interpreter context of each of the Apache worker processes. This is to avoid those issues with some third party packages for Python with C extensions as mentioned before.

In this scenario, to set the location of the Python virtual environment to be used, the `WSGIPythonHome` directive is used:

```
WSGIPythonHome /usr/local/envs/myapp
```

Note that if the WSGI application is being setup within the context of an Apache `VirtualHost`, the `WSGIPythonHome` cannot be placed inside of the `VirtualHost`. Instead it must be placed outside of all

VirtualHost definitions. This is because it applies to the whole Apache instance and not just the single VirtualHost.

Embedded Mode (Multiple Applications)

Running multiple applications in embedded mode is also similar to when running multiple WSGI applications in one daemon process group. You still need to ensure each WSGI application runs in its own Python sub interpreter context to avoid potential issues with Python web frameworks that don't allow more than one WSGI application to be using it at the same time in a Python interpreter context.

If mounting each WSGI application explicitly you might be using:

```
WSGIScriptAlias /myapp3 /some/path/project/myapp3.wsgi
WSGIScriptAlias /myapp2 /some/path/project/myapp2.wsgi

WSGIScriptAlias / /some/path/project/myapp1.wsgi

<Directory /some/path/project>
    Require all granted
</Directory>
```

If instead the directory containing the WSGI application script files is being mounted, you might be using:

```
WSGIScriptAlias / /some/path/project/

<Directory /some/path/project>
    Require all granted
</Directory>
```

In this scenario, to set the location of the Python virtual environment to be used by all WSGI application, the WSGIPythonHome directive is used:

```
WSGIPythonHome /usr/local/envs/myapps
```

If the WSGI application is being setup within the context of an Apache VirtualHost, the WSGIPythonHome cannot be placed inside of the VirtualHost. Instead it must be placed outside of all VirtualHost definitions. This is because it applies to the whole Apache instance and not just the single VirtualHost.

If each WSGI application needs its own Python virtual environment, then activation of the Python virtual environment needs to be performed in the WSGI script itself as explained previously for the case of daemon mode being used. The WSGIPythonHome directive should be used to refer to an empty Python virtual environment if needed to ensure that any additional Python packages in the main Python installation don't interfere with what packages are installed in the Python virtual environment for each WSGI application.

Adding Additional Module Directories

The python-home option to WSGIDaemonProcess and the WSGIPythonHome directive are the preferred way of specifying the location of the Python virtual environment to be used. If necessary, activation of the Python virtual environment can also be performed from the WSGI script file itself.

If you need to add additional directories to search for Python packages or modules this can also be done. You may want to do this where you need to specify where the actual WSGI application is located, where a WSGI script file needs to import application specific modules.

If you are using daemon mode and want to add additional directories to the Python module search path, you can use the python-path option to WSGIDaemonProcess:

```
WSGIDaemonProcess myapp python-path=/some/path/project
```

This option would be in addition to the `python-home` option used to specify where the Python virtual environment is located.

If you are using embedded mode, you can use the `WSGIPythonPath` directive:

```
WSGIPythonPath /some/path/project
```

This directive is in addition to the `WSGIPythonHome` directive used to specify where the Python virtual environment is located.

In either case, if you need to specify more than one directory, they can be separated using a ':' character.

If you are having to activate the Python virtual enviromment from within a WSGI script and need to add additional directories to the Python module search path, you should modify `sys.path` directly from the WSGI script file.

Note that prior practice was that these ways of setting the Python module search path were used to specify the location of the Python virtual environment. Specifically, they were used to add the `site-packages` directory of the Python virtual environment. You should not do that.

The better way to specify the location of the Python virtual environment is using the `python-home` option of the `WSGIDaemonProcess` directive for daemon mode, or the `WSGIPythonHome` directive for embedded mode. These ways of specifying the Python virtual environment have been available since mod_wsgi 3.0 and Linux distributions have not shipped such an old version of mod_wsgi for quite some time. If you are using the older way, please update your configurations.

Access Control Mechanisms

This document contains information about mechanisms available in mod_wsgi for controlling who can access a WSGI application. This includes coverage of support for HTTP Basic and Digest authentication mechanisms, as well as server side mechanisms for authorisation and host access control.

HTTP User Authentication

The HTTP protocol supports user authentication mechanisms for clients through the 'Authorization' header. The two main examples for this are the Basic and Digest authentication mechanisms.

Unlike other HTTP headers, the authorisation header is not passed through to a WSGI application by default. This is the case as doing so could leak information about passwords through to a WSGI application which should not be able to see them when Apache is performing authentication.

If Apache is performing authentication, a WSGI application can still find out what type of authentication scheme was used by checking the variable `AUTH_TYPE` of the WSGI application environment. The login name of the authorised user can be determined by checking the variable `REMOTE_USER`.

If it is desired that the WSGI application be responsible for handling user authentication, then it is necessary to explicitly configure mod_wsgi to pass the required headers through to the application. This can be done by specifying the WSGIPassAuthorization directive in the appropriate context and setting it to 'On'. Note that prior to mod_wsgi version 2.0c5, this directive could not be used in .htaccess files.

When passing of authorisation information is enabled, the authorisation headers are passed through to a WSGI application in the `HTTP_AUTHORIZATION` variable of the WSGI application environment when the equivalent HTTP request header is present. You will still need to provide your own code to process the header and perform the required hand shaking with the client to indicate whether the client is permitted access.

Apache Authentication Provider

When Apache 2.2 was released, it introduced the concept of authentication providers. That is, Apache implements the hand shaking with the client for authentication mechanisms such as Basic and Digest. All that the user server side code needs to provide is a means of authenticating the actual credentials of the user trying to gain access to the site.

This greatly simplified the implementation of client authentication as the hand shaking for a particular authentication mechanism was implemented only once in Apache and it wasn't necessary for each authentication module to duplicate it. This was particularly good for the Digest authentication mechanism which was non trivial to implement correctly.

Using mod_wsgi 2.0 or later, it is possible using the WSGIAuthUserScript directive to define a Python script file containing code which performs the authenticating of user credentials as outlined.

The required Apache configuration for defining the authentication provider for Basic authentication when using Apache 2.2 would be:

```
AuthType Basic
AuthName "Top Secret"
AuthBasicProvider wsgi
WSGIAuthUserScript /usr/local/wsgi/scripts/auth.wsgi
Require valid-user
```

The 'auth.wsgi' script would then need to contain a 'check_password()' function with a sample as shown below:

```
def check_password(environ, user, password):
    if user == 'spy':
        if password == 'secret':
            return True
        return False
    return None
```

This function should validate that the user exists in the user database and that the password is correct. If the user does not exist at all, then the result should be 'None'. If the user does exist, the result should be 'True' or 'False' depending on whether the password was valid.

If wishing to use Digest authentication, the configuration for Apache 2.2 would instead be:

```
AuthType Digest
AuthName "Top Secret"
AuthDigestProvider wsgi
WSGIAuthUserScript /usr/local/wsgi/scripts/auth.wsgi
Require valid-user
```

The name of the required authentication function for Digest authentication is 'get_realm_hash()'. The result of the function must be 'None' if the user doesn't exist, or a hash string encoding the user name, authentication realm and password:

```
import md5

def get_realm_hash(environ, user, realm):
    if user == 'spy':
        value = md5.new()
        # user:realm:password
        value.update('%s:%s:%s' % (user, realm, 'secret'))
        hash = value.hexdigest()
        return hash
    return None
```

By default the auth providers are executed in context of first interpreter created by Python, ie., '%{GLOBAL}' and always in the Apache child processes, never in a daemon process. The interpreter can be overridden using the 'application-group' option to the script directive. The namespace for authentication groups is shared with that for application groups defined by WSGIApplicationGroup.

Because the auth provider is always run in the Apache child processes and never in the context of a mod_wsgi daemon process, if the authentication check is making use of the internals of some Python web framework, it is recommended that the application using that web framework also be run in embedded mode and the same application group. This is the case as the Python web frameworks often bring in a huge amount of code even if using only one small part of them. This will result in a lot of memory being used in the Apache child processes just to support the auth provider.

If mod_authn_alias is being loaded into Apache, then an aliased auth %rovider can also be defined:

```
<AuthnProviderAlias wsgi django>
WSGIAuthUserScript /usr/local/django/mysite/apache/auth.wsgi \
 application-group=django
</AuthnProviderAlias>

WSGIScriptAlias / /usr/local/django/mysite/apache/django.wsgi

<Directory /usr/local/django/mysite/apache>
Order deny,allow
Allow from all

WSGIApplicationGroup django

AuthType Basic
AuthName "Django Site"
AuthBasicProvider django
Require valid-user
</Directory>
```

An authentication script for Django might then be something like:

```
import os, sys
sys.path.append('/usr/local/django')
os.environ['DJANGO_SETTINGS_MODULE'] = 'mysite.settings'

from django.contrib.auth.models import User
from django import db

def check_password(environ, user, password):
    db.reset_queries()

    kwargs = {'username': user, 'is_active': True}

    try:
        try:
            user = User.objects.get(**kwargs)
        except User.DoesNotExist:
            return None

        if user.check_password(password):
            return True
        else:
            return False
    finally:
        db.connection.close()
```

For both Basic and Digest authentication providers, the 'environ' dictionary passed as first argument is a cut down version of what would be supplied to the actual WSGI application. This includes the 'wsgi.errors' object for the purposes of logging error messages associated with the request.

Any configuration defined by !SetEnv directives is not passed in the 'environ' dictionary because doing so would allow users to override the configuration specified in such a way from a '.htaccess' file. Configuration should as a result be placed into the script file itself.

Although authentication providers were a new feature in Apache 2.2, the mod_wsgi module emulates the functionality so that the above can also be used with Apache 2.0. In using Apache 2.0, the required Apache configuration is however slightly different and needs to be:

```
AuthType Basic
AuthName "Top Secret"
WSGIAuthUserScript /usr/local/wsgi/scripts/auth.wsgi
AuthAuthoritative Off
Require valid-user
```

When using Apache 2.0 however, only support for Basic authentication mechanism is provided. It is not possible to use Digest authentication. When using Apache 1.3, this feature is not available at all.

The benefit of using the Apache authentication provider mechanism rather than the WSGI application doing it all itself, is that it can be used to control access to a number of WSGI applications at the same time as well as static files or dynamic pages implemented by other Apache modules using other programming languages such as PHP or Perl. The mechanism could even be used to control access to CGI scripts.

Apache Group Authorisation

As compliment to the authentication provider mechanism, mod_wsgi 2.0 also provides a mechanism for implementing group authorisation using the Apache 'Require' directive. To use this in conjunction with an inbuilt Apache authentication provider such as a password file, the following Apache configuration would be used:

```
AuthType Basic
AuthName "Top Secret"
AuthBasicProvider dbm
AuthDBMUserFile /usr/local/wsgi/accounts.dbm
WSGIAuthGroupScript /usr/local/wsgi/scripts/auth.wsgi
Require group secret-agents
Require valid-user
```

The 'auth.wsgi' script would then need to contain a 'groups_for_user()' function with a sample as shown below:

```
def groups_for_user(environ, user):
    if user == 'spy':
        return ['secret-agents']
    return ['']
```

The function should supply a list of groups the user is a member of or an empty list otherwise.

The feature may be used with any authentication provider, including one defined using WSGIAuthUserScript.

The 'environ' dictionary passed as first argument is a cut down version of what would be supplied to the actual WSGI application. This includes the 'wsgi.errors' object for the purposes of logging error messages associated with the request.

Any configuration defined by !SetEnv directives is not passed in the 'environ' dictionary because doing so would allow users to override the configuration specified in such a way from a '.htaccess' file. Configuration should as a result be placed into the script file itself.

Configuration of group authorisation is the same whether Apache 2.0 or 2.2 is used. The feature is not available when using Apache 1.3.

By default the group authorisation code is always executed in the context of the first interpreter created by Python, ie., '%{GLOBAL}', and always in the Apache child processes, never in a daemon process. The interpreter can be overridden using the 'application-group' option to the script directive.

Host Access Controls

The authentication provider and group authorisation features help to control access based on the identity of a user. Using mod_wsgi 2.0 it is also possible to limit access based on the machine which the client is connecting from. The path to the script is defined using the WSGIAccessScript directive:

```
WSGIAccessScript /usr/local/wsgi/script/access.wsgi
```

The name of the function that must exist in the script file is 'allow_access()'. It must return True or False:

```
def allow_access(environ, host):
    return host in ['localhost', '::1']
```

The 'environ' dictionary passed as first argument is a cut down version of what would be supplied to the actual WSGI application. This includes the 'wsgi.errors' object for the purposes of logging error messages associated with the request.

Any configuration defined by !SetEnv directives is not passed in the 'environ' dictionary because doing so would allow users to override the configuration specified in such a way from a '.htaccess' file. Configuration should as a result be placed into the script file itself.

By default the access checking code is executed in context of the first interpreter created by Python, ie., '%{GLOBAL}', and always in the Apache child processes, never in a daemon process. The interpreter used can be overridden using the 'application-group' option to the script directive.

File Wrapper Extension

The WSGI specification supports an optional feature that can be implemented by WSGI adapters for platform specific file handling.

- http://www.python.org/dev/peps/pep-0333/#optional-platform-specific-file-handling

What this allows is for a WSGI application to return a special object type which wraps a Python file like object. If that file like object statisfies certain conditions as dictated by a specific platform, then the WSGI adapter is allowed to return the content of that file in an optimised manner.

The intent of this is to provide better performance for serving up static file content than a pure Python WSGI application may itself be able to achieve.

Do note however that for the best performance, static files should always be served by a web server. In the case of mod_wsgi this means by Apache itself rather than mod_wsgi or the WSGI application. Using the web server may not always be possible however, such as for files generated on demand.

Example Of Wrapper Usage

A WSGI adapter implementing this extension needs to supply a special callable object under the key 'wsgi.file_wrapper' in the 'environ' dictionary passed to the WSGI application.

What this callable does will be specific to a WSGI adapter, but it must be a callable that accepts one required positional parameter, and one optional positional parameter. The first parameter is the file like object to be sent, and the second parameter is an optional block size. If the block size is not supplied then the WSGI adapter would choose a value which would be most appropriate for the specific hosting mechanism.

Whatever the WSGI adapter does, the result of the callable must be an iterable object which can be used directly as the response from the WSGI application or for passing into any WSGI middleware. Provided the response content isn't consumed by any WSGI middleware and the iterable object gets passed through the WSGI adapter, the WSGI adapter should recognise the special iterable object and trigger any special handling to return the response in a more efficient way.

Because the support of this platform specific file handling is optional for any specific WSGI adapter, any user code should be coded so as to be able to cope with it not existing.

Using the snippet as described in the WSGI specification as guide, the WSGI application would be written as follows:

```
def application(environ, start_response):
    status = '200 OK'
    response_headers = [('Content-type', 'text/plain')]
    start_response(status, response_headers)

    filelike = file('usr/share/dict/words', 'rb')
    block_size = 4096

    if 'wsgi.file_wrapper' in environ:
            return environ['wsgi.file_wrapper'](filelike, block_size)
    else:
        return iter(lambda: filelike.read(block_size), '')
```

Note that the file must always be opened in binary mode. If this isn't done then on platforms which do CR/LF translation automatically then the original content will not be returned but the translated form. As well as it not being the original content this can cause problems with calculated content lengths if the 'Content-Length' response header is returned by the WSGI application and it has been generated by looking at the actual file size rather than the translated content.

Addition Of Content Length

The WSGI specification does not say anything specific about whether a WSGI adapter should generate a 'Content-Length' response header when the 'wsgi.file_wrapper' extension is used and the WSGI application does not return one itself.

For mod_wsgi at least, if the WSGI application doesn't provide a 'Content-Length' response header it will calculate the response content length automatically as being from the current file position to the end of the file. A 'Content-Length' header will then be added to the response for that value.

As far as is known, only mod_wsgi automatically supplies a 'Content-Length' response header in this way. If consistent behaviour is required on all platforms, the WSGI application should always calculate the length and add the header itself.

Existing Content Length

Where a 'Content-Length' is specified by the WSGI application, mod_wsgi will honour that content length. That is, mod_wsgi will only return as many bytes of the file as specified by the 'Content-Length' header.

This is not a requirement of the WSGI specification, but then this is one area of the WSGI specification which is arguably broken. This manifests in the WSGI specification where it says:

> “”“transmission should begin at the current position within the “file” at the time that transmission begins, and continue until the end is reached”“”

If this interpretation is used, where a WSGI application supplies a ‘Content-Length’ header and the number of bytes listed is less than the number of bytes remaining in the file from the current position, then more bytes than specified by the ‘Content-Length’ header would be returned.

To do this would technically be in violation of HTTP specifications which should dictate that the number of bytes returned be the same as that specified by the ‘Content-Length’ response header if supplied.

Not only is this statement in the WSGI specification arguably wrong, the example snippet of code which shows how to implement a fallback where the ‘wsgi.file_wrapper’ is not present, ie.:

```
if 'wsgi.file_wrapper' in environ:
    return environ['wsgi.file_wrapper'](filelike, block_size)
else:
    return iter(lambda: filelike.read(block_size), '')
```

is also wrong. This is because it doesn’t restrict the amount of bytes returned to that specified by ‘Content-Length’.

Although mod_wsgi for normal iterable content would also discard any bytes in excess of the specified ‘Content-Length’, many other WSGI adapters are not known to do this and would just pass back all content regardless. The result of returning excessive content above the specified ‘Content-Length’ would be the failure of subsequent connections were the connection using keep alive and was pipe lining requests.

This problem is also compounded by the WSGI specification not placing any requirement on WSGI middleware to respect the ‘Content-Length’ response header when processing response content. Thus WSGI middleware could also in general generate incorrect response content by virtue of not honouring the ‘Content-Length’ response header.

Overall, although mod_wsgi does what is the logical and right thing to do, if you need to write code which is portable to other WSGI hosting mechanisms, you should never produce a ‘Content-Length’ response header which lists a number of bytes different to that which would be yielded from an iterable object such as a file like object. Thus it would be impossible to use any platform specific file handling features to return a range of bytes from a file.

Restrictions On Optimisations

Although mod_wsgi always supplies the ‘wsgi.file_wrapper’ callable object as part of the WSGI ‘environ’ dictionary, optimised methods of returning the file contents as the response are not always used.

A general restriction is that the file like object must supply both a ‘fileno()’ and ‘tell()’ method. This is necessary in order to get access to the underlying file descriptor and to determine the current position within the file.

The file descriptor is needed so as to be able to use the ‘sendfile()’ function to return file contents in a more optimal manner. The ‘tell()’ method is needed to be able to calculate response ‘Content-Length’ and to validate that where the WSGI application supplies its own ‘Content-Length’ header that there is sufficient bytes in the file.

Because the ‘sendfile()’ function is used by Apache to return file contents in a more optimal manner and because on Windows a Python file object only provides a Windows file handle and not a file descriptor, no optimisations are available on the Windows platform.

The optimisations are also not able to be used if using Apache 1.3. This is because Apache doesn’t provide access to a mechanism for optimised sending of file contents to a content handler under Apache 1.3.

Finally, optimisations are not used where the WSGI application is running in daemon mode. This is currently disabled because some UNIX platforms do not appear to support use of the ‘sendfile()’ function over UNIX sockets and only support INET sockets. This situation may possibly have changed with recent versions of Linux at least but this has yet to be investigated properly.

Whether or not optimisations are supported, the mod_wsgi ‘wsgi.file_wrapper’ extension generally still performs better than if a pure Python iterable object was used to yield the file contents.

Note that this all presumes that the iterable object returned by 'wsgi.file_wrapper' is actually passed back to mod_wsgi and is not consumed by a WSGI middleware. For example, a WSGI middleware which compresses the response content would consume the response content and modify it with a different iterable object being returned. In this case there is no chance for optimisations to be used for returning the file contents.

This problem isn't restricted though to just where the response content is modified in some way and also extends to any WSGI middleware that wants to replace the 'close()' method to perform some cleanup actions at the end of a request.

This is because in order to interject the cleanup actions triggered on the 'close()' method of the iterable object it has to replace the existing iterable object with another which wraps the first, with the outer providing its own 'close()' method. An example of a middleware which replaces the 'close()' method in this way can be found in *Registering Cleanup Code*.

It is thus quite easy for a WSGI application stack to inadvertantly defeat completely any attempts to return file contents in an optimised way using the 'wsgi.file_wrapper' extension of WSGI. As such, attempts should always be used instead to make use of a real web server, whether that be a separate web server, or in the case of mod_wsgi the underlying Apache web server.

Where necessary, features of web servers or proxies such as 'X-Accel-Redirect', 'X-Sendfile' or other special purpose headers could be used. If using mod_wsgi daemon mode and using mod_wsgi version 3.0 or later, the 'Location' response header can also be used.

Registering Cleanup Code

This document describes how to go about registering callbacks to perform cleanup tasks at the end of a request and when an application process is being shutdown.

Cleanup At End Of Request

To perform a cleanup task at the end of a request a couple of different approaches can be used dependent on the requirements. The first approach entails wrapping the calling of a WSGI application within a Python 'try' block, with the cleanup code being triggered from the 'finally' block:

```
def _application(environ, start_response):
    status = '200 OK'
    output = b'Hello World!'

    response_headers = [('Content-type', 'text/plain'),
                        ('Content-Length', str(len(output)))]
    start_response(status, response_headers)

    return [output]

def application(environ, start_response):
    try:
        return _application(environ, start_response)
    finally:
        # Perform required cleanup task.
        ...
```

This might even be factored into a convenient WSGI middleware component:

```
class ExecuteOnCompletion1:
    def __init__(self, application, callback):
```

```
        self.__application = application
        self.__callback = callback
    def __call__(self, environ, start_response):
        try:
            return self.__application(environ, start_response)
        finally:
            self.__callback(environ)
```

The WSGI environment passed in the 'environ' argument to the application could even be supplied to the cleanup callback as shown in case it needed to look at any configuration information or information passed back in the environment from the application.

The application would then be replaced with an instance of this class initialised with a reference to the original application and a suitable cleanup function:

```
def cleanup(environ):
    # Perform required cleanup task.
    ...

application = ExecuteOnCompletion1(_application, cleanup)
```

Using this approach, the cleanup function will actually be called prior to the response content being consumed by mod_wsgi and written back to the client. As such, it is probably only suitable where a complete response is returned as an array of strings. It would not be suitable where a generator is being returned as the cleanup would be called prior to any strings being consumed from the generator. This would be problematic where the cleanup task was to close or delete some resource from which the generator was obtaining the response content.

In order to have the cleanup task only executed after the complete response has been consumed, it would be necessary to wrap the result of the application within an instance of a purpose built generator like object. This object needs to yield each item from the response in turn, and when this object is cleaned up by virtue of the 'close()' method being called, it should in turn call 'close()' on the result returned from the application if necessary, and then call the supplied cleanup callback:

```
class Generator2:
    def __init__(self, iterable, callback, environ):
        self.__iterable = iterable
        self.__callback = callback
        self.__environ = environ
    def __iter__(self):
        for item in self.__iterable:
            yield item
    def close(self):
        try:
            if hasattr(self.__iterable, 'close'):
                self.__iterable.close()
        finally:
            self.__callback(self.__environ)

class ExecuteOnCompletion2:
    def __init__(self, application, callback):
        self.__application = application
        self.__callback = callback
    def __call__(self, environ, start_response):
        try:
            result = self.__application(environ, start_response)
        except:
            self.__callback(environ)
```

```
            raise
        return Generator2(result, self.__callback, environ)
```

Note that for a successfully completed request, since the cleanup task will be executed after the complete response has been written back to the client, if an error occurs there will be no evidence of this in the response seen by the client. As far as the client will be concerned everything will look okay. The only indication of an error will be found in the Apache error log.

Both of the solutions above are not specific to mod_wsgi and should work with any WSGI hosting solution which complies with the WSGI specification.

Cleanup On Process Shutdown

To perform a cleanup task on shutdown of either an Apache child process when using 'embedded' mode of mod_wsgi, or of a daemon process when using 'daemon' mode of mod_wsgi, the standard Python 'atexit' module can be used:

```
import atexit

def cleanup():
    # Perform required cleanup task.
    ...

atexit.register(cleanup)
```

Such a registered cleanup function will also be called if the 'Interpreter' reload mechanism is enabled and the Python sub interpreter in which the cleanup function was registered was destroyed.

Note that although mod_wsgi will ensure that cleanup functions registered using the 'atexit' module will be called correctly, this solution may not be portable to all WSGI hosting solutions.

Also be aware that although one can register a cleanup function to be called on process shutdown, this is no absolute guarantee that it will be called. This is because a process may crash, or it may be forcibly killed off by Apache if it takes too long to shutdown normally. As a result, an application should not be dependent on cleanup functions being called on process shutdown and an application must have some means of detecting an abnormal shutdown when it is started up and recover from it automatically.

Assorted Tips And Tricks

This document contains various tips and tricks related to using mod_wsgi which don't deserve a document of their own or which don't fit within other documentation.

Determining If Running Under mod_wsgi

As a WSGI application developer you should always be striving to write portable WSGI applications. That is, you should not write your code so as to be dependent on the specific features of a specific WSGI hosting mechanism.

This unfortunately is not always possible especially when it comes to deployment due to there being no one blessed way for exposing a WSGI application for hooking into WSGI hosting mechanisms. There may also be times when you might want to rely on a feature of a specific WSGI hosting mechanism, which although not part of the WSGI specification, allows you to do something you wouldn't otherwise.

That said, there a few ways in which you can detect that your code is running under mod_wsgi. These fall under two categories. The first being a general mechanism for how to detect if mod_wsgi is being used. The second being additional ways to detect that mod_wsgi is being used when a request is being handled.

The simplest way of detecting if mod_wsgi is being used is to import the 'mod_wsgi' module. This is a special embedded mode which is installed automatically by the Apache/mod_wsgi module into set of imported modules, ie., sys.modules. You can thus do:

```
try:
    import mod_wsgi
    # Put code here which should only run when mod_wsgi is being used.
except:
    pass
```

Do note however that although this is an embedded mode added automatically, the way mod_wsgi has been implemented allows in the future for there to be a separate Python package/module distinct from the mod_wsgi.so file called 'mod_wsgi' which might contain additional Python code to support use of mod_wsgi.

What would happen if such a separate Python package/module is available is that it will be automatically imported and additional information setup by the Apache/mod_wsgi module then inserted into the global namespace of that Python package/module.

The potential existance of this distinct Python package/module means that importing 'mod_wsgi' could one day actually succeed outside of code being run under the Apache/mod_wsgi module.

A more correct test therefore is:

```
try:
    from mod_wsgi import version
    # Put code here which should only run when mod_wsgi is being used.
except:
    pass
```

This is different because the 'version' attribute will only be present when running under the Apache/mod_wsgi module as that version relates to the version of mod_wsgi.so.

The above import check can be used anywhere, be that in the WSGI script file, or in your application code at either global scope or within the context of a specific function.

In the specific case of the WSGI script file, although the above can be used there is an alternate check that can be made. That is to check the value of the '__name__' attribute given to the WSGI script file when the code is loaded into the Python interpreter.

The normal situation where one would check the value of '__name__' is where wanting to do something different when a Python code file is executed directly against the Python interpreter as opposed to being imported. For example:

```
if __name__ == '__main__':
    ...
```

In contrast, were a Python code file is imported, the '__name__' attribute would be the dotted path which would be used to import the code file.

In the case of mod_wsgi, although WSGI script files are imported as if they are a module, because they could exist anywhere and not in locations on the Python module search path, they don't have a conventional dotted path name. Instead they have a magic name built from a md5 hash of the path to the WSGI script file.

So as to at least identify this as being related to mod_wsgi, it has the prefix '_mod_wsgi_'. This means a WSGI script file could use:

```
if __name__.startswith('_mod_wsgi_'):
    ...
```

if it needed to execute different code based on whether the WSGI script file was actually being loaded by the Apache/mod_wsgi module as opposed to be executed directly as a script by the command line Python interpreter.

This latter technique obviously only works in the WSGI script file and not elsewhere.

A final method that can be used within the context of the WSGI application handling the request is to interrogate the WSGI environ dictionary passed to the WSGI application. In this case code can look for the presence of the 'mod_wsgi.version' key within the WSGI environ dictionary:

```
def application(environ, start_response):
    status = '200 OK'
    if environ.has_key('mod_wsgi.version'):
        output = b'Hello mod_wsgi!'
    else:
        output = b'Hello other WSGI hosting mechanism!'

    response_headers = [('Content-type', 'text/plain'),
                        ('Content-Length', str(len(output)))]
    start_response(status, response_headers)

    return [output]
```

Issues With Pickle Module

This article describes various limitations on what data can be stored using the "pickle" module from a WSGI application script file. This arises due to the fact that a WSGI application script file is not treated exactly the same as a standard Python module.

Note that these limitations only apply to the WSGI application script file which is the target of the WSGIScriptAlias, AddHandler or Action directives. Any standard Python modules or packages which make up an application and which are being imported from directories located in `sys.path` using the 'import' statement are not affected.

Packing And Script Reloading

The first source of problems and limitations is how the operation of the "pickle" serialisation routine is affected by the ability of mod_wsgi to automatically reload WSGI application script files. The particular types of data which are known to be affected are function objects and class objects.

To illustrate the problems and where they arise, consider the following output from an interactive Python session:

```
>>> import pickle
>>> def a(): pass
...
>>> pickle.dumps(a)
'c__main__\na\np0\n.'
>>> z = a
>>> pickle.dumps(z)
'c__main__\na\np0\n.'
```

As can be seen, it is possible to pickle a function object. This can be done even through a copy of the function object by reference, although in that case the pickled object still refers to the original function object.

If now the original function object is deleted however, and the copy of the function object is pickled, a failure will occur:

```
>>> del a
>>> pickle.dumps(z)
... <deleted>
```

```
pickle.PicklingError: Can't pickle <function a at 0x612b0>: it's not found as __main__
↪.a
Traceback (most recent call last):
```

The exception has been raised because the original function object was deleted from where it was created. It occurs because the copy of the original function object is still internally identified by the name which it was assigned at the point of creation. The "pickle" serialisation routine will check that the original object as identified by the name still exists. If it doesn't exist, it will refuse to serialise the object.

Creating a new function object in place of the original function object does not eliminate the problem, although it does result in a different sort of exception:

```
>>> def a(): pass
...
>>> pickle.dumps(z)
... <deleted>
pickle.PicklingError: Can't pickle <function a at 0x612b0>: it's not the same object␣
↪as __main__.a
Traceback (most recent call last):
```

In this case, the "pickle" serialisation routine recognises that "a" exists but realises that it is actually a different function object from which the "z" copy was originally made.

Where the problems start occuring with mod_wsgi is if the function object being saved was itself a copy of some function object which is held outside of the module the function object was defined in. If the module holding the original function object was actually the WSGI application script file and it was reloaded because of the automatic script reloading mechanism, an attempt to pickle the object will fail. This is because the original function object which had been copied from will have been replaced by a new one when the script was reloaded.

This sort of problem, although it will not occur for an instance of a class, will occur for the class object itself:

```
>>> class B: pass
...
>>> b=B()
>>> pickle.dumps(b)
'(i__main__\nB\np0\n(dp1\nb.'
>>> del B
>>> pickle.dumps(b)
'(i__main__\nB\np0\n(dp1\nb.'
>>> class B: pass
...
>>> pickle.dumps(B)
'c__main__\nB\np0\n.'
>>> C = B
>>> pickle.dumps(C)
'c__main__\nB\np0\n.'
>>> del B
>>> pickle.dumps(C)
... <deleted>
pickle.PicklingError: Can't pickle <class __main__.B at 0x53ab0>: it's not found as __
↪main__.B
Traceback (most recent call last):
```

Note though that for the case of a class instance, an appropriate class object must exist at the same location when the serialised object is being restored:

```
>>> class B: pass
...
```

```
>>> b = B()
>>> pickle.loads(pickle.dumps(b))
<__main__.B instance at 0x41e40>
>>> del B
>>> pickle.loads(pickle.dumps(b))
... <delete>
AttributeError: 'module' object has no attribute 'B'
Traceback (most recent call last):
```

Unpacking And Module Names

The second problem derives from how the mod_wsgi script loading mechanism does not make use of the standard Python module importing mechanism. This is necessary as the standard Python module importing mechanism requires every loaded module to have a unique name, with each module residing in `sys.modules` under that name. Further, that name must be able to be used to import the module.

The mod_wsgi script loading mechanism does not place modules in `sys.modules` under their original name so as to allow multiple modules with the same name in different directories and also to avoid having to use the ".py" extension for script files.

The consequence though of modules not residing in `sys.modules` under their original name is that function objects and class objects within such a module may not be able to converted back into objects from their serialised form. This is because "pickle" when attempting to import a module automatically if the module isn't already loaded will not be able to load the WSGI application script file.

The problem can be seen in the following output from an interactive Python session:

```
>>> exec "class C: pass" in m.__dict__
>>> c = m.C()
>>> pickle.dumps(c)
'(im\nC\np0\n(dp1\nb.'
>>> pickle.loads(pickle.dumps(c))
<m.C instance at 0x9a0d0>
>>> del sys.modules["m"]
>>> pickle.loads(pickle.dumps(c))
... <deleted>
ImportError: No module named m
Traceback (most recent call last):
```

Summary Of Limitations

Although the first problem described above could be avoided by disabling script reloading, there is no way to work around the second problem resulting from how mod_wsgi names modules when stored in `sys.modules`.

In practice, what this means is that neither function objects, class objects or instances of classes which are defined in a WSGI application script file should be stored using the "pickle" module.

In order to ensure that no strange problems at all are likely to occur, it is suggested that only basic builtin Python types, ie., scalars, tuples, lists and dictionaries, be stored using the "pickle" module from a WSGI application script file. That is, avoid any type of object which has user defined code associated with it.

Note that this limitation only applies to the WSGI application script file, it doesn't apply to normal Python modules imported using the Python "import" statement.

Issues With Expat Library

This article describes problems caused due to mismatches in the version of the "expat" library embedded into Python and that linked into Apache. Where incompatible versions are used, Apache can crash as soon as any Python code module imports the "pyexpat" module.

Note that this only applies to Python versions prior to Python 2.5. From Python 2.5 onwards, the copy of the "expat" library bundled in with Python is name space prefixed, thereby avoid name clashes with an "expat" library which has previously been loaded.

The Dreaded Segmentation Fault

When moving beyond creating simple WSGI applications to more complicated tasks, one can unexpectedly be confronted with Apache crashing. This generally manifests in no response being returned to the browser when a request is made. Upon further investigation of the Apache error log file, a message similar to the following message is found:

```
[notice] child pid 3238 exit signal Segmentation fault (11)
```

The change which causes this is the explicit addition of code to import the Python module "pyexpat", or the importing of any Python module which indirectly makes use of the "pyexpat" module. Examples of other modules which make use of the "pyexpat" module are "xmlrpclib" and modules from the "PyXML" package. Nearly always, any module which in some way performs processing of XML data will be affected as most such modules rely on using the "pyexpat" module in some way.

Verifying Expat Is The Problem

To verify that the "pyexpat" module is the trigger for the problem, construct a simple WSGI application script file containing:

```
def application(environ, start_response):
    status = '200 OK'
    output = 'without expat\n'

    response_headers = [('Content-type', 'text/plain'),
                        ('Content-Length', str(len(output)))]
    start_response(status, response_headers)

    return [output]
```

Verify that this handler works and the browser receives the response "without pyepxat". Now modify the handler such that the "pyexpat" module is being imported. Also change the response so that it is clear that the modified handler is being used:

```
import pyexpat

def application(environ, start_response):
    status = '200 OK'
    output = 'with expat\n'

    response_headers = [('Content-type', 'text/plain'),
                        ('Content-Length', str(len(output)))]
    start_response(status, response_headers)

    return [output]
```

Presuming that script reloading is enabled, if now upon a request being received by the WSGI application a succesful response of "with pyexpat" is received by the browser, it would generally indicate that the "pyexpat" module is not the problem after all. If however no response is received and the Apache error log records a "Segmentation fault" then the "pyexpat" module is the trigger.

Mismatch In Versions Of Expat

Segmentation faults can occur with any application where different components of the application were compiled against different versions of a common library such as the "expat" library. The actual cause of the problem is generally a change in the API of the library, such as changed function prototypes, changed data types, or changes in structure layouts. In the case where mod_wsgi is being used, the different components are Apache and the "pyexpat" module from Python.

Normally when different components of an application are built, they would be built against the same version of the library and such problems would not occur. In the case of the "pyexpat" module however, it is compiled against a distinct version of the "expat" library which is then embedded within the "pyexpat" module. At the same time, Apache will be built against the version of the "expat" library included with the operating system, or if not a standard part of the operating system, a version which is supplied with Apache.

Thus if the version of the "expat" library embedded into the "pyexpat" module is different to that which Apache was compiled against, the potential for this problem will exist. Note though that there may not always be a problem. Whether there is or not will ultimately depend on what changes were made in the "expat" library between the releases of the different versions used. It is also possible how each library version was compiled could be a factor.

Expat Version Used By Apache

To determine the version of the the "expat" library which is used by Apache, on Linux the "ldd" command can be used. Other operating systems also provide this program or will generally have some form of equivalent program. For example, on Mac OS X the command which is run is "otool -L".

The purpose of these programs is to generate a list of all shared libraries that an application is linked against. To determine where the "expat" library being used by Apache is located, it is necessary to run the "ldd" program on the "httpd" program. On a Linux system, the "httpd" program is normally located in "/usr/sbin". Because we are only interested in the "expat" library, we can ignore anything but the reference to that library:

```
[grahamd@dscpl grahamd]$ ldd /usr/sbin/httpd | grep expat
        libexpat.so.0 => /usr/lib/libexpat.so.0 (0xb7e8c000)
```

From this output it can be seen that the "httpd" program appears to be using "/usr/lib/libexpat.so.0". Although some operating systems embed in the name of the shared library versioning information, it does not generally indicate the true version of the code base which made up the library. To obtain this, it is necessary to extract the version information out of the library. For the "expat" library this can be determined by searching within the strings contained in the library for a version string starting with `expat_`:

```
[grahamd@dscpl grahamd]$ strings /usr/lib/libexpat.so.0 | grep expat_
expat_1.95.8
```

The version of the "expat" library would therefore appear to be "1.95.8". Unfortunately though, many operating systems allow the library search path to be overridden at the point that a program is run using an environment variable such as "LD_LIBRARY_PATH" and it is quite possible that when Apache is run, the context in which it is run could result in it finding the "expat" library in a different location.

To be absolutely sure, it is necessary to determine which "expat" library the running copy of Apache used. On Linux and many other operating systems, this can be determined using the "lsof" command. If this program doesn't exist, an

alternate program which may be available is "ofiles". Either of these should be run against one of the active Apache processes. If Apache was originally started as root, the command will also need to be run as root:

```
[grahamd@dscpl grahamd]$ ps aux | grep http | head -3
root      3625  0.0  0.6 31068 12836 ?       SN   Sep25   0:08 /usr/sbin/httpd
apache   24814  0.0  0.7 34196 15604 ?       SN   04:11   0:00 /usr/sbin/httpd
apache   24815  0.0  0.7 33924 15916 ?       SN   04:11   0:00 /usr/sbin/httpd

[grahamd@dscpl grahamd]$ sudo /usr/sbin/lsof -p 3625 | grep expat
httpd   3625 root  mem    REG     253,0   123552    6409040
/usr/lib/libexpat.so.0.5.0

[grahamd@dscpl grahamd]$ strings /usr/lib/libexpat.so.0.5.0 | grep expat_
expat_1.95.8
```

Expat Version Used By Python

To determine the version of the "expat" library which is embedded in the Python "pyexpat" module, the module should be imported and the version information extracted from the module. This can be done by executing "python" on the command line and entering the necessary code directly:

```
[grahamd@dscpl grahamd]$ python
Python 2.3.3 (#1, May  7 2004, 10:31:40)
[GCC 3.3.3 20040412 (Red Hat Linux 3.3.3-7)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import pyexpat
>>> pyexpat.version_info
(1, 95, 7)
```

Combining Python And Apache

When mod_wsgi is used from within Apache, although there is a version of the "expat" library embedded in the "pyexpat" module, it will effectively be ignored. This is because Apache has already loaded into memory at startup the version of the "expat" library which it is linked against. That this occurs can be seen by using the ability of Linux to forcibly preload a shared library into a program when run, even though that program wasn't linked against the library orginally. This is achieved using the "LD_PRELOAD" environment variable:

```
[grahamd@dscpl grahamd]$ LD_PRELOAD=/usr/lib/libexpat.so.0.5.0 python
Python 2.3.3 (#1, May  7 2004, 10:31:40)
[GCC 3.3.3 20040412 (Red Hat Linux 3.3.3-7)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import pyexpat
>>> pyexpat.version_info
(1, 95, 8)
```

As can be seen, although the "pyexpat" module for this version of Python embedded version 1.95.7 of the "expat" library, when the same version of the "expat" library as was being used by Apache is forcibly loaded into the program at startup, the version information obtained from the "pyexpat" module now shows that version 1.95.8 of the "expat" library is being used.

Luckily in this case, the patch level difference between the two versions of the "expat" library as used by Python and Apache doesn't cause a problem. If however the two versions of the "expat" library were incompatible, one would expect to see the "python" program crash with a segmentation fault at this point. This therefore can be used as an alternate way of verifying that it is the "pyexpat" module and more specifically the version of the "expat" library used, that is causing the problem.

Updating System Expat Version

Because the version of the "expat" library embedded within the "pyexpat" module is shipped as source code within the Python distribution, it can be hard to replace it. The preferred approach to resolving the mismatch is therefore to replace/update the version of the "expat" library that is used by Apache.

Generally the problem occurs where that used by Apache is older than that which is being used by Python. In that case, the version of the "expat" library used by Apache should be updated to be the same version as that embedded within the "pyexpat" module. By using the same version, one would expect any problems to disappear. If problems still persist, it is possible that Apache may also need to be recompiled against the same version of the "expat" library as used in Python.

CHAPTER 8

Configuration

WSGIAcceptMutex

Description Specify type of accept mutex used by daemon processes.

Syntax `WSGIAcceptMutex Default` | *method*

Default `WSGIAcceptMutex Default`

Context server config

The `WSGIAcceptMutex` directive sets the method that mod_wsgi will use to serialize multiple daemon processes in a process group accepting requests on a socket connection from the Apache child processes. If this directive is not defined then the same type of mutex mechanism as used by Apache for the main Apache child processes when accepting connections from a client will be used. If set the method types are the same as for the Apache AcceptMutex directive.

Note that the `WSGIAcceptMutex` directive and corresponding features are not available on Windows or when running Apache 1.3.

WSGIAccessScript

Description Specify script implementing host access controls.

Syntax `WSGIAccessScript` *path* [*options*]

Context directory, .htaccess

Override AuthConfig

The `WSGIAccessScript` directive provides a mechanism for implementing host access controls.

More detailed information on using the `WSGIAccessScript` directive can be found in *Access Control Mechanisms*.

The options which can be supplied to the `WSGIAccessScript` directive are:

application-group=name

Specifies the name of the application group within the specified process for which the script file will be loaded.

If the `application-group` option is not supplied, the special value `%{GLOBAL}` which denotes that the script file be loaded within the context of the first interpreter created by Python when it is initialised will be used. Otherwise, will be loaded into the interpreter for the specified application group.

Note that the script always runs in processes associated with embedded mode. It is not possible to delegate the script such that it is run within context of a daemon process.

WSGIApplicationGroup

Description Sets which application group WSGI application belongs to.

Syntax `WSGIApplicationGroup name` `WSGIApplicationGroup %{GLOBAL}` `WSGIApplicationGroup %{SERVER}` `WSGIApplicationGroup %{RESOURCE}` `WSGIApplicationGroup %{ENV:variable}`

Default `WSGIApplicationGroup %{RESOURCE}`

Context server config, virtual host, directory

The `WSGIApplicationGroup` directive can be used to specify which application group a WSGI application or set of WSGI applications belongs to. All WSGI applications within the same application group will execute within the context of the same Python sub interpreter of the process handling the request.

The argument to the `WSGIApplicationGroup` can be either one of four special expanding variables or an explicit name of your own choosing. The meaning of the special variables are:

%{GLOBAL}

The application group name will be set to the empty string.

Any WSGI applications in the global application group will always be executed within the context of the first interpreter created by Python when it is initialised. Forcing a WSGI application to run within the first interpreter can be necessary when a third party C extension module for Python has used the simplified threading API for manipulation of the Python GIL and thus will not run correctly within any additional sub interpreters created by Python.

%{SERVER}

The application group name will be set to the server hostname. If the request arrived over a non standard HTTP/HTTPS port, the port number will be added as a suffix to the group name separated by a colon.

For example, if the virtual host `www.example.com` is handling requests on the standard HTTP port (80) and HTTPS port (443), a request arriving on either port would see the application group name being set to `www.example.com`. If instead the virtual host was handling requests on port 8080, then the application group name would be set to `www.example.com:8080`.

%{RESOURCE}

The application group name will be set to the server hostname and port as for the `%{SERVER}` variable, to which the value of WSGI environment variable `SCRIPT_NAME` is appended separated by the file separator character.

For example, if the virtual host `www.example.com` was handling requests on port 8080 and the URL-path which mapped to the WSGI application was:

```
http://www.example.com/wsgi-scripts/foo
```

then the application group name would be set to:

```
www.example.com:8080|/wsgi-scripts/foo
```

The effect of using the `%{RESOURCE}` variable expansion is for each application on any server to be isolated from all others by being mapped to its own Python sub interpreter.

%{ENV:variable}

The application group name will be set to the value of the named environment variable. The environment variable is looked-up via the internal Apache notes and subprocess environment data structures and (if not found there) via `getenv()` from the Apache server process.

In an Apache configuration file, environment variables accessible using the `%{ENV}` variable reference can be setup by using directives such as SetEnv and RewriteRule.

For example, to group all WSGI scripts for a specific user when using mod_userdir within the same application group, the following could be used:

```
RewriteEngine On
RewriteCond %{REQUEST_URI} ^/~([^/]+)
RewriteRule . - [E=APPLICATION_GROUP:~%1]

<Directory /home/*/public_html/wsgi-scripts/>
Options ExecCGI
SetHandler wsgi-script
WSGIApplicationGroup %{ENV:APPLICATION_GROUP}
</Directory>
```

Note that in embedded mode or a multi process daemon process group, there will be an instance of the named sub interpreter in each process. Thus the directive only ensures that request is handled in the named sub interpreter within the process that handles the request. If you need to ensure that requests for a specific user always go back to the exact same sub interpreter, then you will need to use a daemon process group with only a single process, or implement sticky session mechanism across a number of single process daemon process groups.

WSGIAuthGroupScript

Description Specify script implementing group authorisation.

Syntax `WSGIAuthGroupScript` *path* [*options*]

Context directory, .htaccess

Override AuthConfig

The `WSGIAuthGroupScript` directive provides a mechanism for implementing group authorisation using the Apache `Require` directive.

More detailed information on using the `WSGIAuthGroupScript` directive can be found in *Access Control Mechanisms*.

The options which can be supplied to the `WSGIAuthGroupScript` directive are:

application-group=name

Specifies the name of the application group within the specified process for which the script file will be loaded.

If the `application-group` option is not supplied, the special value `%{GLOBAL}` which denotes that the script file be loaded within the context of the first interpreter created by Python when it is initialised will be used. Otherwise, will be loaded into the interpreter for the specified application group.

Note that the script always runs in processes associated with embedded mode. It is not possible to delegate the script such that it is run within context of a daemon process.

WSGIAuthUserScript

Description Specify script implementing an authentication provider.

Syntax `WSGIAuthUserScript` *path* [*options*]

Context directory, .htaccess

Override AuthConfig

The WSGIAuthUserScript directive can be used to specify a script which implements an Apache authentication provider.

Such an authentication provider can be used where you want Apache to worry about the handshaking related to HTTP Basic and Digest authentication and you only wish to deal with supplying the user credentials for authenticating the user.

If using at least Apache 2.2, other Apache modules implementing custom authentication mechanisms can also make use of the authentication provider if they are using the corresponding Apache C API for accessing them.

More detailed information on using the WSGIAuthUserScript directive can be found in *Access Control Mechanisms*.

The options which can be supplied to the WSGIAuthUserScript directive are:

application-group=name Specifies the name of the application group within the specified process for which the script file will be loaded.

If the 'application-group' option is not supplied, the special value '%{GLOBAL}' which denotes that the script file be loaded within the context of the first interpreter created by Python when it is initialised will be used. Otherwise, will be loaded into the interpreter for the specified application group.

Note that the script always runs in processes associated with embedded mode. It is not possible to delegate the script such that it is run within context of a daemon process.

WSGICallableObject

Description Sets the name of the WSGI application callable.

Syntax `WSGICallableObject` *name* `WSGICallableObject %{ENV:variable}`

Default `WSGICallableObject application`

Context server config, virtual host, directory, .htaccess

Override `FileInfo`

The WSGICallableObject directive can be used to override the name of the Python callable object in the script file which is used as the entry point into the WSGI application.

When `%{ENV}` is being used, the environment variable is looked-up via the internal Apache notes and subprocess environment data structures and (if not found there) via getenv() from the Apache server process.

In an Apache configuration file, environment variables accessible using the `%{ENV}` variable reference can be setup by using directives such as SetEnv and RewriteRule.

Note that the name of the callable object must be an object present at global scope within the WSGI script file. It is not possible to use a dotted path to refer to a sub object of a module imported by the WSGI script file.

WSGICaseSensitivity

Description Define whether file system is case sensitive.

Syntax `WSGICaseSensitivity On|Off`

Context server config

When mod_wsgi is used on the Windows and MacOS X platforms, it will assume that the filesystem in use is case insensitive. This is necessary to ensure that the module caching system works correctly and only one module is retained in memory where paths with different case are used to identify the same script file. On other platforms it will always be assumed that a case sensitive file system is used.

The WSGICaseSensitivity directive can be used explicitly to specify for a particular WSGI application whether the file system the script file is stored in is case sensitive or not, thus overriding the default for any platform. A value of On indicates that the filesystem is case sensitive.

Because it is set in the main server config it will apply to the whole site. All paths therefore would need to be located in a filesystem with the same case convention.

WSGIDaemonProcess

Description Configure a distinct daemon process for running applications.

Syntax `WSGIDaemonProcess` *name* `[` *options* `]`

Context server config, virtual host

The `WSGIDaemonProcess` directive can be used to specify that distinct daemon processes should be created to which the running of WSGI applications can be delegated. Where Apache has been started as the `root` user, the daemon processes can be run as a user different to that which the Apache child processes would normally be run as.

When distinct daemon processes are enabled and used, the process is dedicated to mod_wsgi and the only thing that the processes do is run the WSGI applications assigned to that process group. Any other Apache modules such as PHP or activities such as serving up static files continue to be run in the standard Apache child processes.

Note that having denoted that daemon processes should be created by using the `WSGIDaemonProcess` directive, the `WSGIProcessGroup` directive, or the `process-group` option of `WSGIScriptAlias` still needs to be used to delegate specific WSGI applications to execute within those daemon processes.

Also note that the name of the daemon process group must be unique for the whole server. That is, it is not possible to use the same daemon process group name in different virtual hosts.

Options which can be supplied to the `WSGIDaemonProcess` directive are:

processes=num Defines the number of daemon processes that should be started in this process group. If not defined then only one process will be run in this process group.

Note that if this option is defined as `processes=1`, then the WSGI environment attribute called `wsgi.multiprocess` will be set to be `True` whereas not providing the option at all will result in the attribute being set to be `False`. This distinction is to allow for where some form of load balancing is used across

process groups in the same Apache instance, or separate Apache instances. If you need to ensure that `wsgi.multiprocess` is `False` so that interactive debuggers will work, simply do not specify the `processes` option and allow the default single daemon process to be created in the process group.

threads=num Defines the number of threads to be created to handle requests in each daemon process within the process group.

If this option is not defined then the default will be to create 15 threads in each daemon process within the process group.

Do not get carried away and set this to a very large number in the belief that it will somehow magically enable you to handle many more concurrent users. Any sort of increased value would only be appropriate where your code is I/O bound. If you code is CPU bound, you are better of using at most 3 to 5 threads per process and using more processes.

display-name=value Defines a different name to show for the daemon process when using the `ps` command to list processes. If the value is `%{GROUP}` then the name will be `(wsgi:group)` where `group` is replaced with the name of the daemon process group.

Note that only as many characters of the supplied value can be displayed as were originally taken up by `argv0` of the executing process. Anything in excess of this will be truncated.

This feature may not work as described on all platforms. Typically it also requires a `ps` program with BSD heritage. Thus on some versions of Solaris UNIX the `/usr/bin/ps` program doesn't work, but `/usr/ucb/ps` does. Other programs which can display this value include `htop`.

home=directory Defines an absolute path of a directory which should be used as the initial current working directory of the daemon processes within the process group.

If this option is not defined the initial current working directory will be set to be the home directory of the user that the daemon process is configured to run as using the `user` option to the `WSGIDaemonProcess` directive. Otherwise the current working directory of Apache when started will be used, which if Apache is being started from system init scripts, would usually be the system root directory.

user=name | user=#uid Defines the UNIX user *name* or numeric user *uid* of the user that the daemon processes should be run as. If this option is not supplied the daemon processes will be run as the same user that Apache would run child processes, as defined by the User directive, and it is not necessary to set this to the Apache user yourself.

Note that this option is ignored if Apache wasn't started as the root user, in which case no matter what the settings, the daemon processes will be run as the user that Apache was started as.

Also be aware that mod_wsgi will not allow you to run a daemon process group as the root user due to the security risk of running a web application as root.

group=name | group=#gid Defines the UNIX group *name* or numeric group *gid* of the primary group that the daemon processes should be run as. If this option is not supplied the daemon processes will be run as the same group that Apache would run child processes, as defined by the Group directive, and it is not necessary to set this to the Apache group yourself.

Note that this option is ignored if Apache wasn't started as the root user, in which case no matter what the settings, the daemon processes will be run as the group that Apache was started as.

supplementary-groups=group1 | supplementary-groups=group1,group2 Defines a list of additional UNIX groups that the user the daemon process group runs as, should be added to, in addition to primary UNIX group associated with that user. When specifying more than one group, separate the names of the groups with a comma.

umask=0nnn Defines a value to be used for the umask of the daemon processes within the process group. The value must be provided as an octal number.

If this option is not defined then the umask of the user that Apache is initially started as will be inherited by the process. Typically the inherited umask would be '0022'.

lang=locale Set the current language locale. This is the same as having set the `LANG` environment variable.

You will need to set this on many Linux systems where Apache when started up from system init scripts uses the default C locale, meaning that the default system encoding is ASCII. Unless you need a special language locale, set this to `en_US.UTF-8`.

Whether the `lang` or `locale` option works best can depend on the system being used. Set both if you aren't sure which is appropriate.

locale=locale Set the current language locale. This is the same as having set the `LC_ALL` environment variable.

You will need to set this on many Linux systems where Apache when started up from system init scripts uses the default C locale, meaning that the default system encoding is ASCII. Unless you need a special language locale, set this to `en_US.UTF-8`.

Whether the `lang` or `locale` option works best can depend on the system being used. Set both if you aren't sure which is appropriate.

chroot=directory Run the daemon process group process within a chroot jail. Use of a chroot jail is now deprecated due to the difficulty in setting up a chroot environment. It is recommended that you use more modern containerisation technologies such as Docker or runC.

script-user=name | script-user=#uid Sets the user that must be the owner of any WSGI script file delegated to be run in the daemon process group. If the owner doesn't match a HTTP Forbidden response will be returned for any request.

Note that this doesn't change what user the daemon process group runs as at any time. If you want to set the user that the daemon process group runs as, use the `user` option.

Only one of `script-user` or `script-group` option can be used at the same time.

script-group=name | scrip-group=#gid Sets the group that must be the group of any WSGI script file delegated to be run in the daemon process group. If the group doesn't match a HTTP Forbidden response will be returned for any request.

Note that this doesn't change what group the daemon process group runs as at any time. If you want to set the group that the daemon process group runs as, use the `group` option.

Only one of `script-user` or `script-group` option can be used at the same time.

python-home=directory Set the location of the Python virtual environment to be used by the daemon processes. The directory to use is that which `sys.prefix` is set to for the Python virtual environment. The virtual environment can have been created by `virtualenv`, `pyvenv` or `python -m venv`.

Note that the Python virtual environment must have been created using the same base Python version as was used to compile the mod_wsgi module. You can't use this to force mod_wsgi to somehow use a different Python version than it was compiled for. If you want to use a different version of Python, you will need to reinstall mod_wsgi, compiling it for the version you want. It is not possible for the one mod_wsgi instance to run applications for both Python 2 and 3 at the same time.

python-path=directory | python-path=directory:directory List of colon separated directories to add to the Python module search path, ie., `sys.path`.

Note that this is not strictly the same as having set the `PYTHONPATH` environment variable when running normal command line Python. When this option is used, the directories are added by calling `site.addsitedir()`. As well as adding the directory to `sys.path` this function has the effect of opening and interpreting any `.pth` files located in the specified directories.

If using a Python virtual environment, rather than use this option to refer to the `site-packages` directory of the Python virtual environment, you should use the `python-home` option to specify the root of the Python virtual environment instead.

In all cases, if the directory contains Python packages which have C extension components, those packages must have been installed using the same base Python version as was used to compile the mod_wsgi module. You should not mix packages from different Python versions or installations.

python-eggs=directory Directory to be used as the Python egg cache directory. This is equivalent to having set the `PYTHON_EGG_CACHE` environment variable.

Note that the directory specified must exist and be writable by the user that the daemon process run as.

restart-interval=nnn Defines a time limit on how long a daemon process should run before being restarted.

This might be use to periodically force restart the WSGI application processes when you have issues related to Python object reference count cycles, or incorrect use of in memory caching, which causes constant memory growth.

If this option is not defined, or is defined to be 0, then the daemon process will be persistent and will continue to service requests until Apache itself is restarted or shutdown.

Avoid setting this too low. This is because the constant restarting and reloading of your WSGI application may cause unecessary load on your system and affect performance.

You can use the `graceful-timeout` option in conjunction with this option to reduce the chances that an active request will be interrupted when a restart occurs due to the use of this option.

maximum-requests=nnn Defines a limit on the number of requests a daemon process should process before it is shutdown and restarted.

This might be use to periodically force restart the WSGI application processes when you have issues related to Python object reference count cycles, or incorrect use of in memory caching, which causes constant memory growth.

If this option is not defined, or is defined to be 0, then the daemon process will be persistent and will continue to service requests until Apache itself is restarted or shutdown.

Avoid setting this to a low number of requests on a site which handles a lot of traffic. This is because the constant restarting and reloading of your WSGI application may cause unecessary load on your system and affect performance. Only use this option if you have no other choice due to a memory usage issue. Stop using it as soon as any memory issue has been resolved.

You can use the `graceful-timeout` option in conjunction with this option to reduce the chances that an active request will be interrupted when a restart occurs due to the use of this option.

inactivity-timeout=sss Defines the maximum number of seconds allowed to pass before the daemon process is shutdown and restarted when the daemon process has entered an idle state. For the purposes of this option, being idle means there are no currently active requests and no new requests are being received.

This option exists to allow infrequently used applications running in a daemon process to be restarted, thus allowing memory being used to be reclaimed, with process size dropping back to the initial startup size before any application had been loaded or requests processed.

Note that after any restart of the WSGI application process, the WSGI application will need to be reloaded. This can mean that the first request received by a process after the process was restarted can be slower. If you WSGI application has a very high startup cost on CPU and time, it may not be a good idea to use the option.

See also the `request-timeout` option for forcing a process restart when requests block for a specified period of time.

Note that similar functionality to that of the `request-timeout` option, for forcing a restart when requests blocked, was part of what was implemented by the `inactivity-timeout` option. The request timeout was broken out into a separate feature in version 4.1.0 of mod_wsgi.

request-timeout=sss Defines the maximum number of seconds that a request is allowed to run before the daemon process is restarted. This can be used to recover from a scenario where a request blocks indefinitely, and where

if all request threads were consumed in this way, would result in the whole WSGI application process being blocked.

How this option is seen to behave is different depending on whether a daemon process uses only one thread, or more than one thread for handling requests, as set by the `threads` option.

If there is only a single thread, and so the process can only handle one request at a time, as soon as the timeout has passed, a restart of the process will be initiated.

If there is more than one thread, the request timeout is applied to the average running time for any requests, across all threads. This means that a request can run longer than the request timeout. This is done to reduce the possibility of interupting other running requests, and causing a user to see a failure. So where there is still capacity to handle more requests, restarting of the process will be delayed if possible.

deadlock-timeout=sss Defines the maximum number of seconds allowed to pass before the daemon process is shutdown and restarted after a potential deadlock on the Python GIL has been detected. The default is 300 seconds.

This option exists to combat the problem of a daemon process freezing as the result of a rouge Python C extension module which doesn't properly release the Python GIL when entering into a blocking or long running operation.

startup-timeout=sss Defines the maximum number of seconds allowed to pass waiting to see if a WSGI script file can be loaded successfully by a daemon process. When the timeout is passed, the process will be restarted.

This can be used to force the reloading of a process when a transient issue occurs on the first attempt to load the WSGI script file, but subsequent attempts still fail because a Python package that was loaded has retained state that prevents attempts to run initialisation a second time within the same process. The Django package can cause this scenario as the initialisation of Django itself can no longer be attempted more than once in the same process.

graceful-timeout=sss When `maximum-requests` is used and the maximum has been reached, or `cpu-time-limit` is used and the CPU limit reached, or `restart-interval` is used and the time limit reached, if `graceful-timeout` is set, then the process will continue to run for the number of second specified by this option, while still accepting new requests, to see if the process reaches an idle state. If the process reaches an idle state, it will then be resarted immediately. If the process doesn't reach an idle state and the graceful restart timeout expires, the process will be restarted, even if it means that requests may be interrupted.

eviction-timeout=sss When a daemon process is sent the graceful restart signal, usually `SIGUSR1`, to restart a process, this timeout controls how many seconds the process will wait, while still accepting new requests, before it reaches an idle state with no active requests and shutdown.

If this timeout is not specified, then the value of the `graceful-timeout` will instead be used. If the `graceful-timeout` is not specified, then the restart when sent the graceful restart signal will instead happen immediately, with the process being forcibly killed, if necessary, when the shutdown timeout has expired.

shutdown-timeout=sss Defines the maximum number of seconds allowed to pass when waiting for a daemon process to shutdown. When this timeout has been reached the daemon process will be forced to exited even if there are still active requests or it is still running Python exit functions. The shutdown timeout is applied after any graceful restart timeout or eviction timeout if they have been specified. No new requests are accepted during the shutdown timeout is being applied.

If this option is not defined, then the shutdown timeout will be set to 5 seconds. Note that this option does not change the shutdown timeout applied to daemon processes when Apache itself is being stopped or restarted. That timeout value is defined internally to Apache as 3 seconds and cannot be overridden.

connect-timeout=sss Defines the maximum amount of time for an Apache child process to wait trying to get a successful connection to the mod_wsgi daemon processes. This defaults to 15 seconds.

socket-timeout=sss Defines the timeout on individual reads/writes on the socket connection between the Apache child processes and the mod_wsgi daemon processes. If this is not specified, the number of seconds specified by the Apache Timeout directive will be used instead.

queue-timeout=sss Defines the timeout on how long to wait for a mod_wsgi daemon process to accept a request for processing.

This option is to allow one to control what to do when backlogging of requests occurs. If the daemon process is overloaded and getting behind, then it is more than likely that a user will have given up on the request anyway if they have to wait too long. This option allows you to specify that a request that was queued up waiting for too long is discarded, allowing any transient backlog to be quickly discarded and not simply cause the daemon process to become even more backlogged. When this occurs the user will recieve a 504 Gateway Time Out response.

listen-backlog=nnn Defines the depth of the daemon process socket listener queue. By default the limit is 100, although this is actually a hint, as different operating systems can have different limits on the maximum value or otherwise treat it in special ways.a

This option can be set, along with `queue-timeout` to try and better handle back logging when the WGSI application gets overloaded.

socket-user=name | socket-user=#uid Set the owner of the UNIX listener socket for the daemon process group.

This can be used when using the Apache PrivilegesMode directive with value of `SECURE` to change the owner of the socket from the default Apache user, to the user under which the Apache child process which is attempting to connect to the daemon process group, will run when handling requests. This is necessary otherwise the Apache child worker process will not be able to connect to the listener socket for the mod_wsgi daemon process to proxy the request to the WSGI application.

This option can also be used when using third party Apache modules such as mod_ruid, mod_ruid2, mod_suid as well as the ITK MPM for Apache.

cpu-time-limit=sss Define the maximum amount of CPU time a daemon process is allowed to consume before a shutdown is triggered and the daemon process restarted. The point of this is to provide some means of controlling potentially run away processes due to bad code that gets stuck in heavy processing loops.

Note that CPU time used is recorded from when the daemon process is first created. This means that a process will eventually reach the limit in normal use and would be restarted. You can use the `graceful-timeout` option to reduce the chances that an active request will be interrupted.

cpu-priority=num Sets the scheduling priority set to the daemon processes. This can be a number of the range -20 to 20. The default priority is 0. A lower priority gives more favourable scheduling.

memory-limit=num Sets the maximum amount of memory a daemon process can use. This will have no affect on some platforms as `RLIMIT_AS`/`RLIMIT_DATA` with `setrlimit()` isn't always implemented. For example MacOS X and older Linux kernel versions do not implement this feature. You will need to test whether this feature works or not before depending on it.

virtual-memory-limit=num Sets the maximum amount of virtual memory a daemon process can use. This will have no affect on some platforms as `RLIMIT_VMEM` with `setrlimit()` isn't always implemented. You will need to test whether this feature works or not before depending on it.

stack-size=nnn The amount of virtual memory in bytes to be allocated for the stack corresponding to each thread created by mod_wsgi in a daemon process.

This option would be used when running Linux in a VPS system which has been configured with a quite low 'Memory Limit' in relation to the 'Context RSS' and 'Max RSS Memory' limits. In particular, the default stack size for threads under Linux is 8MB is quite excessive and could for such a VPS result in the 'Memory Limit' being exceeded before the RSS limits were exceeded. In this situation, the stack size should be dropped down to be in the region of 512KB (524288 bytes).

receive-buffer-size=nnn Defines the UNIX socket buffer size for data being received by the daemon process from the Apache child process.

This option may need to be used to override small default values set by certain operating systems and would help avoid possibility of deadlock between Apache child process and daemon process when the WSGI application generates large responses but doesn't consume request content. In general such deadlock problems would not arise with well behaved WSGI applications, but some spam bots attempting to post data to web sites are known to trigger the problem.

The maximum possible value that can be set for the buffer size is operating system dependent and will need to be calculated through trial and error.

send-buffer-size=nnn Defines the UNIX socket buffer size for data being sent in the direction daemon process back to Apache child process.

This option may need to be used to override small default values set by certain operating systems and would help avoid possibility of deadlock between Apache child process and daemon process when the WSGI application generates large responses but doesn't consume request content. In general such deadlock problems would not arise with well behaved WSGI applications, but some spam bots attempting to post data to web sites are known to trigger the problem.

The maximum possible value that can be set for the buffer size is operating system dependent and will need to be calculated through trial and error.

header-buffer-size=nnn Defines the maximum size that a response header/value can be that is returned from a WSGI application. The default size is 32768 bytes. This might need to be overridden where excessively large response headers are returned, such as in custom authentication challenge schemes which use the `WWW-Authenticate` header.

response-buffer-size=nnn Defines the maximum number of bytes that will be buffered for a response in the Apache child processes when proxying the response body from the WSGI application. The default size is 65536 bytes. Be careful increasing this to provide extra buffering of responses as it contributes to the runtime memory size of the Apache child processes.

response-socket-timeout=nnn Defines the maximum number of seconds allowed to pass before timing out on a write operation back to the HTTP client when the response buffer has filled and data is being forcibly flushed. Defaults to 0 seconds indicating that it will default to the value of the `socket-timeout` option.

To delegate a particular WSGI application to run in a named set of daemon processes, the `WSGIProcessGroup` directive should be specified in appropriate context for that application, or the `process-group` option used on the `WSGIScriptAlias` directive. If neither is used to delegate the WSGI application to run in a daemon process group, the application will be run within the standard Apache child processes.

If the `WSGIDaemonProcess` directive is specified outside of all virtual host containers, any WSGI application can be delegated to be run within that daemon process group. If the `WSGIDaemonProcess` directive is specified within a virtual host container, only WSGI applications associated with virtual hosts with the same server name as that virtual host can be delegated to that set of daemon processes.

In the case where you have two separate `VirtualHost` definitions for the same `ServerName`, but where one is for port 80 and the other for port 443, specify the `WSGIDaemonProcess` directive in the first `VirtualHost`. You can then refer to that daemon process group by name from the second `VirtualHost`. Using one daemon process group across the two virtual hosts in this case is preferred as then you do not have two whole separate instances of your application for port 80 and 443.

```
<VirtualHost *:80>
ServerName www.site1.com

WSGIDaemonProcess www.site1.com user=joe group=joe processes=2 threads=25
WSGIProcessGroup www.site1.com

...
</VirtualHost>
```

```
<VirtualHost *:443>
ServerName www.site1.com

WSGIProcessGroup www.site1.com

...
</VirtualHost>
```

When `WSGIDaemonProcess` is associated with a virtual host, the error log associated with that virtual host will be used for all Apache error log output from mod_wsgi rather than it appear in the main Apache error log.

For example, if a server is hosting two virtual hosts and it is desired that the WSGI applications related to each virtual host run in distinct processes of their own and as a user which is the owner of that virtual host, the following could be used:

```
<VirtualHost *:80>
ServerName www.site1.com
CustomLog logs/www.site1.com-access_log common
ErrorLog logs/ww.site1.com-error_log

WSGIDaemonProcess www.site1.com user=joe group=joe processes=2 threads=25
WSGIProcessGroup www.site1.com

...
</VirtualHost>

<VirtualHost *:80>
ServerName www.site2.com
CustomLog logs/www.site2.com-access_log common
ErrorLog logs/www.site2.com-error_log

WSGIDaemonProcess www.site2.com user=bob group=bob processes=2 threads=25
WSGIProcessGroup www.site2.com

...
</VirtualHost>
```

Note that the `WSGIDaemonProcess` directive and corresponding features are not available on Windows.

WSGIImportScript

Description Specify a script file to be loaded on process start.

Syntax `WSGIImportScript` *path* `[` *options* `]`

Context server config

The WSGIImportScript directive can be used to specify a script file to be loaded when a process starts. Options must be provided to indicate the name of the process group and the application group into which the script will be loaded.

The options which must supplied to the WSGIImportScript directive are:

process-group=name Specifies the name of the process group for which the script file will be loaded.

The name of the process group can be set to the special value '%{GLOBAL}' which denotes that the script file be loaded for the Apache child processes. Any other value indicates appropriate process group for mod_wsgi daemon mode.

application-group=name Specifies the name of the application group within the specified process for which the script file will be loaded.

The name of the application group can be set to the special value '%{GLOBAL}' which denotes that the script file be loaded within the context of the first interpreter created by Python when it is initialised. Otherwise, will be loaded into the interpreter for the specified application group.

Because the script files are loaded prior to beginning to accept any requests, any delay in loading the script will not cause actual requests to be blocked. As such, the WSGIImportScript can be used to preload a WSGI application script file on process start so that it is ready when actual user requests arrive. For where there are multiple processes handling requests, this can reduce or eliminate the apparent stalling of an application when performing a restart of Apache or a daemon mode process group.

WSGILazyInitialization

Description Enable/disable lazy initialisation of Python.

Syntax `WSGILazyInitialization On|Off`

Default `WSGILazyInitialization On`

Context server config

The WSGILazyInitialization directives sets whether or not the Python interpreter is preinitialised within the Apache parent process or whether lazy initialisation is performed, and the Python interpreter only initialised in the Apache server processes or mod_wsgi daemon processes after they have forked from the Apache parent process.

In versions of mod_wsgi prior to version 3.0 the Python interpreter was always preinitialised in the Apache parent process. This did mean that theoretically some benefit in memory usage could be derived from delayed copy on write semantics of memory inherited by child processes that was initialised in the parent. This memory wasn't significant however and was tempered by the fact that the Python interpreter when destroyed and then reinitialised in the Apache parent process on an Apache restart, would with some Python versions leak memory. This meant that if a server had many restarts performed, the Apache parent process and thus all forked child processes could grow in memory usage over time, eventually necessitating Apache be completely stopped and then restarted.

This issue of memory leaks with the Python interpreter reached an extreme with Python 3.0, where by design, various data structures would not be destroyed on the basis that it would be reused when Python interpreter was reinitialised within the same process. The problem is that when an Apache restart is performed, mod_wsgi and the Python library are unloaded from memory, with the result that the references to that memory would be lost and so a real memory leak, of significant size and much worse that older versions of Python, would result.

As a consequence, with mod_wsgi 3.0 and onwards, the Python interpreter is not initialised by default in the Apache parent process for any version of Python. This avoids completely the risk of cummulative memory leaks by the Python interpreter on a restart into the Apache parent process, albeit with potential for a slight increase in child process memory sizes. If need be, the existing behaviour can be restored by setting the directive with the value 'Off'.

A further upside of using lazy initialisation is that if you are using daemon mode only, ie., not using embedded mode, you can completely turn off initialisation of the Python interpreter within the main Apache server child process. Unfortunately, because it isn't possible in the general case to know whether embedded mode will be needed or not, you will need to manually set the configuration to do this. This can be done by setting:

```
WSGIRestrictEmbedded On
```

With restrictions on embedded mode enabled, any attempt to run a WSGI application in embedded mode will fail, so it will be necessary to ensure all WSGI applications are delegated to run in daemon mode. Although WSGI applications will be restricted from being run in embedded mode and the Python interpreter therefore not initialised, it will fallback to being initialised if you use any of the Python hooks for access control, authentication or authorisation providers, or WSGI application dispatch overrides.

Note that if mod_python is being used in the same Apache installation, because mod_python takes precedence over mod_wsgi in initialising the Python interpreter, lazy initialisation cannot be done and so Python interpreter will continue to be preinitialised in the Apache parent process regardless of the setting of WSGILazyInitialization. Use of mod_python will thus perpetuate the risk of memory leaks and growing memory use of Apache process. This is especially the case since mod_python doesn't even properly destroy the Python interpreter in the Apache parent process on a restart and so all memory associated with the Python interpreter is leaked and not just that caused by the Python interpreter when it is destroyed and doesn't clean up after itself.

WSGIPassAuthorization

Description Enable/Disable passing of authorisation headers.

Syntax `WSGIPassAuthorization On|Off`

Default `WSGIPassAuthorization Off`

Context server config, virtual host, directory, .htaccess

The WSGIPassAuthorization directive can be used to control whether HTTP authorisation headers are passed through to a WSGI application in the `HTTP_AUTHORIZATION` variable of the WSGI application environment when the equivalent HTTP request headers are present. This option would need to be set to `On` if the WSGI application was to handle authorisation rather than Apache doing it.

Authorisation headers are not passed through by default as doing so could leak information about passwords through to a WSGI application which should not be able to see them when Apache is performing authorisation. If Apache is performing authorisation, a WSGI application can still find out what type of authorisation scheme was used by checking the variable `AUTH_TYPE` of the WSGI application environment. The login name of the authorised user can be determined by checking the variable `REMOTE_USER`.

WSGIProcessGroup

Description Sets which process group WSGI application is assigned to.

Syntax `WSGIProcessGroup %{GLOBAL}|%{ENV:variable}|name`

Default `WSGIProcessGroup %{GLOBAL}`

Context server config, virtual host, directory

The WSGIProcessGroup directive can be used to specify which process group a WSGI application or set of WSGI applications will be executed in. All WSGI applications within the same process group will execute within the context of the same group of daemon processes.

The argument to the WSGIProcessGroup can be either one of two special expanding variables or the actual name of a group of daemon processes setup using the WSGIDaemonProcess directive. The meaning of the special variables are:

%{GLOBAL} The process group name will be set to the empty string.

Any WSGI applications in the global process group will always be executed within the context of the standard Apache child processes. Such WSGI applications will incur the least runtime overhead, however, they will share the same process space with other Apache modules such as PHP, as well as the process being used to serve up static file content. Running WSGI applications within the standard Apache child processes will also mean the application will run as the user that Apache would normally run as.

%{ENV:variable} The process group name will be set to the value of the named environment variable. The environment variable is looked-up via the internal Apache notes and subprocess environment data structures and (if not

found there) via getenv() from the Apache server process. The result must identify a named process group setup using the WSGIDaemonProcess directive.

In an Apache configuration file, environment variables accessible using the `%{ENV}` variable reference can be setup by using directives such as SetEnv and RewriteRule.

For example, to select which process group a specific WSGI application should execute within based on entries in a database file, the following could be used:

```
RewriteEngine On
RewriteMap wsgiprocmap dbm:/etc/httpd/wsgiprocmap.dbm
RewriteRule . - [E=PROCESS_GROUP:${wsgiprocmap:%{REQUEST_URI}}]

WSGIProcessGroup %{ENV:PROCESS_GROUP}
```

When using the WSGIProcessGroup directive, only daemon process groups defined within virtual hosts with the same server name, or those defined at global scope outside of any virtual hosts can be selected. It is not possible to select a daemon process group which is defined within a different virtual host. Which daemon process groups can be selected may be further restricted if the WSGIRestrictProcess directive has been used.

Note that the WSGIProcessGroup directive and corresponding features are not available on Windows or when running Apache 1.3.

WSGIPythonEggs

Description Directory to use for Python eggs cache.

Syntax `WSGIPythonEggs` *directory*

Context server config

Used to specify the directory to be used as the Python eggs cache directory for all sub interpreters created within embedded mode. This directive achieves the same affect as having set the `PYTHON_EGG_CACHE` environment variable.

Note that the directory specified must exist and be writable by the user that the Apache child processes run as. The directive only applies to mod_wsgi embedded mode. To set the Python eggs cache directory for mod_wsgi daemon processes, use the 'python-eggs' option to the WSGIDaemonProcess directive instead.

WSGIPythonHome

Description Absolute path to Python prefix/exec_prefix directories.

Syntax `WSGIPythonHome` *prefix|prefix:exec_prefix*

Context server config

Used to indicate to Python when it is initialised where its library files are installed. This should be defined where the Python executable is not in the `PATH` of the user that Apache runs as, or where a system has multiple versions of Python installed in different locations in the file system, especially different installations of the same major/minor version, and the installation that Apache finds in its `PATH` is not the desired one.

This directive can also be used to indicate a Python virtual environment created using a tool such as `virtualenv`, to be used for the whole of mod_wsgi.

When this directive is used it should be supplied the prefix for the directories containing the platform independent and system dependent Python library files. The directories should be separated by a ':'. If the same directory is used for

both, then only the one directory path needs to be supplied. Where the directories are the same, this can usually be determined by looking at the value of the `sys.prefix` variable for the version of Python being used.

Note that the Python installation being referred to using this directive must be the same major/minor version of Python that mod_wsgi was compiled for. If you want to use a different version of major/minor version of Python than currently used, you must recompile mod_wsgi against the alternate version of Python.

This directive is the same as having set the environment variable `PYTHONHOME` in the environment of the user that Apache executes as. If this directive is used it will override any setting of `PYTHONHOME` in the environment of the user that Apache executes as.

This directive will have no affect if mod_python is being loaded into Apache at the same time as mod_wsgi as mod_python will in that case be responsible for initialising Python.

This directive is not available on Windows systems. Note that mod_wsgi 1.X will not actually reject this directive if listed in the configuration, however, it also will not do anything either. This is because on Windows systems Python ignores the `PYTHONHOME` environment variable and always seems to use the location of the Python DLL for determining where the library files are located.

WSGIPythonOptimize

Description Enables basic Python optimisation features.

Syntax `WSGIPythonOptimize [0|1|2]`

Default `WSGIPythonOptimize 0`

Context server config

Sets the level of Python compiler optimisations. The default is '0' which means no optimisations are applied.

Setting the optimisation level to '1' or above will have the effect of enabling basic Python optimisations and changes the filename extension for compiled (bytecode) files from `.pyc` to `.pyo`.

On the Windows platform, optimisation level of '0' apparently results in the same outcome as if the optimisation level had been set to '1'.

When the optimisation level is set to '2', doc strings will not be generated and thus not retained. This may techically result in a smaller memory footprint if all `.pyo` files were compiled at this optimisation level, but may cause some Python packages which interrogate doc strings in some way to fail.

Since all the installed `.pyo` files in your Python installation are not likely to be installed with level '2' optimisation, the gain from using this level of optimisation will probably be negligible if any. This is because potentially only the Python code for your own application code will be compiled with this level of optimisation. This will be the case as the `.pyo` files will aready exist for modules in the standard Python library and they will be used as is, rather than them being regenerated with a higher level of optimisation than they might be. Use of level '2' optimisation is therefore discouraged.

This directive will have no affect if mod_python is being loaded into Apache at the same time as mod_wsgi as mod_python will in that case be responsible for initialising Python.

Overall, if you do not understand what the normal 'python' executable `-O` option does, how the Python runtime changes it behaviour as a result, and you don't know exactly how your application would be affected by enabling this option, then do not use this option. In other words, stop trying to prematurely optimise the performance of your application through shortcuts. You will get much better performance gains by looking at the design of your application and eliminating bottlenecks within it and how it uses any database. So, put the gun down and back away, it will be better for all concerned.

WSGIPythonPath

Description Additional directories to search for Python modules.

Syntax `WSGIPythonPath` *directory|directory-1:directory-2:...*

Context server config

Used to specify additional directories to search for Python modules. If multiple directories are specified they should be separated by a ':' if using a UNIX like system, or ';' if using Windows. If any part of a directory path contains a space character, the complete argument string to WSGIPythonPath must be quoted.

When using mod_wsgi version 1.X, this directive is the same as having set the environment variable `PYTHONPATH` in the environment of the user that Apache executes as. If this directive is used it will override any setting of `PYTHONPATH` in the environment of the user that Apache executes as. The end result is that the listed directories will be added to `sys.path`.

Note that in mod_wsgi version 1.X this applies to all Python sub interpreters created, be they in the Apache child processes when embedded mode is used, or in distinct daemon processes when daemon mode is used. It is not possible to define this differently for mod_wsgi daemon processes. If additional directories need to be added to the module search path for a specific WSGI application it should be done within the WSGI application script itself.

When using mod_wsgi version 2.0, this directive does not have the same affect as having set the environment variable `PYTHONPATH`. In fact, if `PYTHONPATH` is set in the environment of the user that Apache is started as, any directories so defined will still be added to `sys.path` and they will not be overridden.

The difference with this directive when using mod_wsgi 2.0 is that each directory listed will be added to the end of `sys.path` by calling `site.addsitedir()`. By using this function, as well as the directory being added to `sys.path`, any '.pth' files located in the directories will be opened and processed. Thus, if the directories contain Python eggs, any associated directories corresponding to those Python eggs will in turn also be added automatically to `sys.path`.

Note however that when using mod_wsgi 2.0, this directive only sets up the additional Python module search directories for interpreters created in the Apache child processes where embedded mode is used. If directories need to be specified for interpreters running in daemon processes, the 'python-path' option to the WSGIDaemonProcess directive corresponding to that daemon process should instead be used.

In mod_wsgi version 2.0, because directories corresponding to Python eggs are automatically added to `sys.path`, the directive can be used to point at the `site-packages` directory corresponding to a Python virtual environment created by a tool such as `virtualenv`.

For mod_wsgi 1.X, this directive will have no affect if mod_python is being loaded into Apache at the same time as mod_wsgi as mod_python will in that case be responsible for initialising Python.

WSGIRestrictEmbedded

Description Enable restrictions on use of embedded mode.

Syntax `WSGIRestrictEmbedded On|Off`

Default `WSGIRestrictEmbedded Off`

Context server config

The WSGIRestrictEmbedded directive determines whether mod_wsgi embedded mode is enabled or not. If set to 'On' and the restriction on embedded mode is therefore enabled, any attempt to make a request against a WSGI application which hasn't been properly configured so as to be delegated to a daemon mode process will fail with a HTTP internal server error response.

This option does not exist on Windows, or Apache 1.3 or any other configuration where daemon mode is not available.

WSGIRestrictProcess

Description Restrict which daemon process groups can be selected.

Syntax `WSGIRestrictProcess` *group-1 group-2 ...*

Syntax WSGIRestrictProcess *group-1 group-2 ...*

Context server config, virtual host, directory

When using the WSGIProcessGroup directive, daemon process groups defined within virtual hosts with the same server name, or those defined at global scope outside of any virtual hosts can be selected. It is not possible to select a daemon process group which is defined within a different virtual host.

To further limit which of the available daemon process groups can be selected, the WSGIRestrictProcess directive can be used to list a restricted set of daemon process group names. This could be used for example where %{ENV} substitution is being used to allow the daemon process group to be selected from a .htaccess file for a specific user.

The main Apache configuration for this scenario might be:

```
WSGIDaemonProcess default processes=2 threads=25

<VirtualHost *:80>
ServerName www.site.com

WSGIDaemonProcess bob:1 user=bob group=bob threads=25
WSGIDaemonProcess bob:2 user=bob group=bob threads=25
WSGIDaemonProcess bob:3 user=bob group=bob threads=25

WSGIDaemonProcess joe:1 user=joe group=joe threads=25
WSGIDaemonProcess joe:2 user=joe group=joe threads=25
WSGIDaemonProcess joe:3 user=joe group=joe threads=25

SetEnv PROCESS_GROUP default
WSGIProcessGroup %{ENV:PROCESS_GROUP}

<Directory /home/bob/public_html>
Options ExecCGI
AllowOverride FileInfo
AddHandler wsgi-script .wsgi
WSGIRestrictProcess bob:1 bob:2 bob:3
SetEnv PROCESS_GROUP bob:1
</Directory>
</VirtualHost>
```

The .htaccess file within the users account could then delegate specific WSGI applications to different daemon process groups using the SetEnv directive:

```
<Files blog.wsgi>
SetEnv PROCESS_GROUP bob:2
</Files>

<Files wiki.wsgi>
SetEnv PROCESS_GROUP bob:3
</Files>
```

Note that the WSGIDaemonProcess directive and corresponding features are not available on Windows or when running Apache 1.3.

WSGIRestrictSignal

Description Enable restrictions on use of signal().

Syntax `WSGIRestrictSignal On|Off`

Default `WSGIRestrictSignal On`

Context server config

A well behaved Python WSGI application should not in general register any signal handlers of its own using `signal.signal()`. The reason for this is that the web server which is hosting a WSGI application will more than likely register signal handlers of its own. If a WSGI application were to override such signal handlers it could interfere with the operation of the web server, preventing actions such as server shutdown and restart.

In the interests of promoting portability of WSGI applications, mod_wsgi restricts use of `signal.signal()` and will ensure that any attempts to register signal handlers are ignored. A warning notice will be output to the Apache error log indicating that this action has been taken.

If for some reason there is a need for a WSGI application to register some special signal handler this behaviour can be turned off, however an application should avoid the signals `SIGTERM`, `SIGINT`, `SIGHUP`, `SIGWINCH` and `SIGUSR1` as these are all used by Apache.

Apache will ensure that the signal `SIGPIPE` is set to `SIG_IGN`. If a WSGI application needs to override this, it must ensure that it is reset to `SIG_IGN` before any Apache code is run. In a multi threaded MPM this would be practically impossible to ensure so it is preferable that the handler for `SIG_PIPE` also not be changed.

Apache does not use `SIGALRM`, but it is generally preferable that other techniques be used to achieve the same affect.

Do note that if enabling the ability to register signal handlers, such a registration can only reliably be done from within code which is implemented as a side effect of importing a script file identified by the WSGIImportScript directive. This is because signal handlers can only be registered from the main Python interpreter thread, and request handlers when using embedded mode and a multithreaded Apache MPM would generally execute from secondary threads. Similarly, when using daemon mode, request handlers would executed from secondary threads. Only code run as a side effect of WSGIImportScript is guaranteed to be executed in main Python interpreter thread.

WSGIRestrictStdin

Description Enable restrictions on use of STDIN.

Syntax `WSGIRestrictStdin On|Off`

Default `WSGIRestrictStdin On`

Context server config

A well behaved Python WSGI application should never attempt to read any input directly from `sys.stdin`. This is because ways of hosting WSGI applications such as CGI use standard input as the mechanism for receiving the content of a request from the web server. If a WSGI application were to directly read from `sys.stdin` it could interfere with the operation of the WSGI adapter and result in corruption of the input stream.

In the interests of promoting portability of WSGI applications, mod_wsgi restricts access to `sys.stdin` and will raise an exception if an attempt is made to use `sys.stdin` explicitly.

The only time that one might want to remove this restriction is if the Apache web server is being run in debug or single process mode for the purposes of being able to run an interactive Python debugger such as `pdb`.

WSGIRestrictStdout

Description Enable restrictions on use of STDOUT.

Syntax `WSGIRestrictStdout On|Off`

Default `WSGIRestrictStdout On`

Context server config

A well behaved Python WSGI application should never attempt to write any data directly to `sys.stdout` or use the `print` statement without directing it to an alternate file object. This is because ways of hosting WSGI applications such as CGI use standard output as the mechanism for sending the content of a response back to the web server. If a WSGI application were to directly write to `sys.stdout` it could interfere with the operation of the WSGI adapter and result in corruption of the output stream.

In the interests of promoting portability of WSGI applications, mod_wsgi restricts access to `sys.stdout` and will raise an exception if an attempt is made to use `sys.stdout` explicitly.

The only time that one might want to remove this restriction is purely out of convencience of being able to use the `print` statement during debugging of an application, or if some third party module or WSGI application was errornously using `print` when it shouldn't. If restrictions on using `sys.stdout` are removed, any data written to it will instead be sent through to `sys.stderr` and will appear in the Apache error log file.

WSGIScriptAlias

Description Maps a URL to a filesystem location and designates the target as a WSGI script.

Syntax `WSGIScriptAlias` *URL-path file-path|directory-path*

Context server config, virtual host

The WSGIScriptAlias directive behaves in the same manner as the Alias directive, except that it additionally marks the target directory as containing WSGI scripts, or marks the specific *file-path* as a script, that should be processed by mod_wsgi's `wsgi-script` handler.

Where the target is a *directory-path*, URLs with a case-sensitive (%-decoded) path beginning with *URL-path* will be mapped to scripts contained in the indicated directory.

For example:

```
WSGIScriptAlias /wsgi-scripts/ /web/wsgi-scripts/
```

A request for `http://www.example.com/wsgi-scripts/name` in this case would cause the server to run the WSGI application defined in `/web/wsgi-scripts/name`. This configuration is essentially equivalent to:

```
Alias /wsgi-scripts/ /web/wsgi-scripts/
<Location /wsgi-scripts>
SetHandler wsgi-script
Options +ExecCGI
</Location>
```

Where the target is a *file-path*, URLs with a case-sensitive (%-decoded) path beginning with *URL-path* will be mapped to the script defined by the *file-path*.

For example:

```
WSGIScriptAlias /name /web/wsgi-scripts/name
```

A request for `http://www.example.com/name` in this case would cause the server to run the WSGI application defined in `/web/wsgi-scripts/name`.

If possible you should avoid placing WSGI scripts under the DocumentRoot in order to avoid accidentally revealing their source code if the configuration is ever changed. The WSGIScriptAlias makes this easy by mapping a URL and designating the location of any WSGI scripts at the same time. If you do choose to place your WSGI scripts in a directory already accessible to clients, do not use WSGIScriptAlias. Instead, use Directory, SetHandler and Options as in:

```
<Directory /usr/local/apache/htdocs/wsgi-scripts>
SetHandler wsgi-script
Options ExecCGI
</Directory>
```

This is necessary since multiple *URL-paths* can map to the same filesystem location, potentially bypassing the WSGIScriptAlias and revealing the source code of the WSGI scripts if they are not restricted by a Directory section.

WSGIScriptAliasMatch

Description Maps a URL to a filesystem location and designates the target as a WSGI script.

Syntax `WSGIScriptAliasMatch` *regex file-path|directory-path*

Context server config, virtual host

This directive is similar to the WSGIScriptAlias directive, but makes use of regular expressions, instead of simple prefix matching. The supplied regular expression is matched against the URL-path, and if it matches, the server will substitute any parenthesized matches into the given string and use it as a filename.

For example, to map a URL to scripts contained within a directory where the script files use the `.wsgi` extension, but it is desired that the extension not appear in the URL, use:

```
WSGIScriptAliasMatch ^/wsgi-scripts/([^/]+) /web/wsgi-scripts/$1.wsgi
```

Note that you should only use WSGIScriptAliasMatch if you know what you are doing. In most cases you should be using WSGIScriptAlias instead. If you use WSGIScriptAliasMatch and don't do things the correct way, then you risk modifying the value of SCRIPT_NAME as passed to the WSGI application and this can stuff things up badly causing URL mapping to not work correctly within the WSGI application or stuff up reconstruction of the full URL when doing redirects. This is because the substitution of the matched sub pattern from the left hand side back into the right hand side is often critical.

If you think you need to use WSGIScriptAliasMatch, you probably don't really. If you really really think you need it, then check on the mod_wsgi mailing list about how to use it properly.

WSGIScriptReloading

Description Enable/Disable detection of WSGI script file changes.

Syntax `WSGIScriptReloading On|Off`

Default `WSGIScriptReloading On`

Context server config, virtual host, directory, .htaccess

Override `FileInfo`

The WSGIScriptReloading directive can be used to control whether changes to WSGI script files trigger the reloading mechanism. By default script reloading is enabled and a change to the WSGI script file will trigger whichever reloading mechanism is appropriate to the mode being used.

WSGISocketPrefix

Description Configure directory to use for daemon sockets.

Syntax `WSGISocketPrefix` *prefix*

Context server config

Defines the directory and name prefix to be used for the UNIX domain sockets used by mod_wsgi to communicate between the Apache child processes and the daemon processes.

If the directive is not defined, the sockets and any related mutex lock files will be placed in the standard Apache runtime directory. This is the same directory that the Apache log files would normally be placed.

For some Linux distributions, restrictive permissions are placed on the standard Apache runtime directory such that the directory is not readable to others. This can cause problems with mod_wsgi because the user that the Apache child processes run as will subsequently not have the required permissions to access the directory to be able to connect to the sockets.

When this occurs, a '503 Service Temporarily Unavailable' error response would be received by the client. To resolve the problem, the WSGISocketPrefix directive should be defined to point at an alternate location. The value may be a location relative to the Apache root directory, or an absolute path.

On systems which restrict access to the standard Apache runtime directory, they normally provide an alternate directory for placing sockets and lock files used by Apache modules. This directory is usually called 'run' and to make use of this directory the WSGISocketPrefix directive would be set as follows:

```
WSGISocketPrefix run/wsgi
```

Note, do not put the sockets in the system temporary working directory. That is, do not go making the prefix '/tmp/wsgi'. The directory should be one that is only writable by 'root' user, or if not starting Apache as 'root', the user that Apache is started as.

Note that the WSGISocketPrefix directive and corresponding features are not available on Windows or when running Apache 1.3.

CHAPTER 9

Finding Help

If after you have gone through all the available documentation you still cannot work out how to do something or can't resolve a problem you are having, use the mod_wsgi mailing list to post your question. The mailing list is hosted by Google Groups at:

- http://groups.google.com/group/modwsgi

You do not need to have a Google email account as Google Groups allows you to register external email addresses as well.

Please use the mailing list in preference to raising a ticket in the issue tracker, unless you are somewhat certain that the problem is a bug in mod_wsgi and not just some environment issue related to your application, any third party packages being used or the operating system. It is much easier to have a discussion on the mailing list than the issue tracker.

The mailing list also has many people participating, or at least reading, so you have people with a broad experience with many third party Python web packages and operating systems who may be able to help.

If the problem is potentially more an issue with a third party package or the operating system rather than mod_wsgi, you might also consider asking on any mailing list related to the third party package instead.

A further option is to ask your question on StackOverflow, if a programming question, or ServerFault, if an administration issue. These sites allow a broad range of questions about many topics with quite a large user base of sometimes knowledgeable people.

A final option you might try is any IRC channels related to any third party package or the more general #wsgi.

Do be aware though that the only forum that is guaranteed to be monitored is the mod_wsgi mailing list. Questions are not gauranteed to be answered on sites such as StackOverflow and ServerFault, on IRC, or mailing lists for other packages. So, it is much preferable to use the mod_wsgi mailing list if you want an informed answer for a mod_wsgi specific question.

As a general rule, if you have never participated in public forums previously to seek answers to technical questions, including about Open Source software, it is highly recommended you have a read of.

- http://www.catb.org/esr/faqs/smart-questions.html

This will help you to ensure you have exhausted all possibilities as to where to find information and try and solve the problem yourself, as well as assist you in framing your question the best way so as to get the best response possible.

Remember that people on the mailing list are volunteering their time to help and don't get paid for answering questions. Thus, it is in your interest not to annoy them too much.

No matter which forum you use, when asking questions, it is always helpful to detail the following:

1. Which version of mod_wsgi you are using and if using a packaged distribution, who provided the distribution.

   If you are not using the latest version, then upgrade first and verify the problem still occurs with the latest version.

2. Which version of Python you are using and if using a packaged distribution, who provided the distribution.
3. Which version of Apache you are using and if using a packaged distribution, who provided the distribution.

   If not using latest version of Apache available, then consider upgrading and trying again. If at all possible, avoid using Apache 2.0 or 2.2. You definitely shouldn't still be using Apache 1.3

4. What operating system you are using.
5. Details on any third party packages being used and what versions of those packages.
6. The mod_wsgi configuration you are using from Apache configuration files.

   In particular you should indicate whether you are using mod_wsgi embedded mode or daemon mode. Also can be helpful to indicate what MPM Apache has been compiled for and whether mod_php or mod_python are being loaded into the same Apache instance.

7. Relevant error messages from the Apache error logs.

   Specifically, don't just quote the single line you think shows the error message. Instead, also show the lines before and after that point. These other lines from the error logs may show supplemental error messages from Apache or mod_wsgi or provide Python traceback information.

CHAPTER 10

Reporting Bugs

If you believe you have uncovered a bug in mod_wsgi code then lodge a bug report on the mod_wsgi issue tracker. The issue tracker is located on GitHub at:

- https://github.com/GrahamDumpleton/mod_wsgi/issues

Before creating a ticket on the issue tracker, please do try and ensure you attempt to resolve issues using the mod_wsgi mailing list first as explained in *Finding Help*. The majority of issues lodged via the issue tracker are not actually bugs in mod_wsgi but due to external factors or simply a failure to read the documentation.

CHAPTER 11

Contributing

The mod_wsgi package is a solo effort by Graham Dumpleton.

The package is developed purely in the author's spare time and is not funded in any way by a company, nor is it developed for a specific companies requirements. In fact the author doesn't even develop it for his own needs. It is developed purely because it represents an interesting technical challenge and not because the author needs it himself to host a significant web site.

How to make a donation

If you use mod_wsgi and wish to show your appreciation, donations can be made via PayPal or an Amazon (USA store only) gift certificate sent to Graham.Dumpleton at gmail dot com.

A suggested formula for how much to donate is:

- If using mod_wsgi for personal use, then consider donating what you would pay for one months worth of a single host used to run your own site.
- If using mod_wsgi for a company web site, then consider donating what you would pay for two months worth of a single host used to run that site.
- If using mod_wsgi as part of a web hosting service which you then charge other people for using, then consider donating what you would pay for three months worth of a single host used to run that site.

In other words, if you feel inclined, donate an amount commensurate with how much benefit you are getting from mod_wsgi. The reference to the cost of hosting is used at it reflects in some way how much you can afford or might be willing to pay for a hosting service yourself.

On that basis, donations might realistically range from $5 up to $150 or more. Obviously where your company spends ridiculous amounts of money on web hosting you can instead elect to donate something more within the range stated above rather than how much you actually spend on web hosting services.

Now for the reality, which is that it is very rare that a company will ever donate any money to an Open Source project. As such, when donations have occassionally been received (which doesn't happen very often), they are from individuals using mod_wsgi themselves.

Some people do openly begrudge Open Source projects soliciting donations, but the amounts received overall are so insignificant in comparison to how much effort is generally put into projects and what a developer would need to survive that anything received is more a symbolic gesture, more than anything else, of ones appreciation.

Given that donations invariably are from individuals, do know that they are accepted with much gratitude and appreciation in return that you are at least, even if companies aren't, trying to help support Open Source projects in some way.

How else can you donate

If you are an author of a book related to Apache, Python, Docker or any other technologies which go into providing web hosting services, then will also happily accept an electronic copy of the book for reference.

Still don't think a monetary contribution is something you would do, you can also simply send a Twitter message to the author expressing your appreciation. You will be surprised how far positive encouragement and appreciation can go with people who work on Open Source projects. This is because in part satisfaction comes from knowing people are benefiting from the work being done. If you never do or say anything, then Open Source developers will never know that you do appreciate the work they do, so don't be quiet when an Open Source project is of value to you, at least say 'Thank You'.

How are donations used

Any monetary donations typically go towards buying clothes, toys, music, books and apps for the authors 2 children. They are therefore used as a special treat for the authors kids.

Source code contributions

You might be thinking, what about source code contributions. Although it would be great for this project to grow to have multiple developers working on the code and documentation, reality is that working inside of Apache and the Python C APIs is quite specialised. It isn't therefore the most attractive of projects in that regard. If however you are keen, then would love to hear from you.

Open Source free loaders

If you are the sort of person who thinks that the Internet exists only to provide you with free stuff and where you think everyone out there exists purely to help you work out your problems, then it may be better that you go use some other WSGI server project.

Even if you don't contribute as described above, if you at least recognise that other people are giving up their time to help you and that you put in some effort yourself to resolve a problem first, and then explain it properly in some detail to others when seeking help, providing answers to any questions asked of you, then you will still be helped.

The worst sort of people, which hopefully you don't want to be one of, are those who simply say something is broken but will not provide sufficient details, thereby forcing other people to waste huge amounts of time dragging out the information required to help you, or having to guess what your problem is.

It is people in this latter category which are becoming a significant drain on the time of developers of Open Source projects and which are a part of why so many Open Source developers are experiencing burnout. So if you are the sort to expect people to help you, complain about things when the problem is really your own unwillingness to learn, and generally give nothing positive in return, even if only encouragement, then don't expect to be helped. Your like has

caused too much damage in the past already to any number of Open Source projects and will not be tolerated here. The mental health of Open Source developers is more important than you are.

CHAPTER 12

Source Code

The source code repository for mod_wsgi is located on GitHub at:

- https://github.com/GrahamDumpleton/mod_wsgi

Downloadable tar balls of the source code can be found at:

- https://github.com/GrahamDumpleton/mod_wsgi/releases

A version of the source code which can be installed using `pip` can also be found on PyPi at:

- https://pypi.python.org/pypi/mod_wsgi

CHAPTER 13

Release Notes

Version 4.5.17

Version 4.5.17 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.5.17

Bugs Fixed

- Addition in `mod_wsgi-express` of `--allow-override` option in 4.5.16 caused `--url-alias` option to break.

Version 4.5.16

Version 4.5.16 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.5.16

Bugs Fixed

- The `WSGIDontWriteBytecode` option wasn't available when using Python 3.3 and later. This feature of Python wasn't in initial Python 3 versions, but when was later added, mod_wsgi was updated to allow it.
- The feature behind the `startup-timeout` option of `WSGIDaemonProcess` was broken by prior fix related to feature in 4.5.10. This meant the option was not resulting in daemon processes being restarted when the WSGI script file could not be loaded successfully by the specified timeout.
- When using `WSGIImportScript`, or `WSGIScriptAlias` with both the `process-group` and `application-group` options, with the intent of preloading a WSGI script file, the ability to reach across to a daemon process defined in a different virtual host with same `ServerName` was always failing and the target daemon process group would be flagged as not accessible when instead it should have been.

New Features

- Added `--allow-override` option to `mod_wsgi-express` to allow use of a `.htaccess` in document root directory and any directories mapped using a URL alias. The argument to the directive should be the directive type which can be overridden in the `.htaccess` file. The option can be used more than once if needing to allow overriding of more than one directive type. Argument can be anything allowed by `AllowOverride` directive.

Version 4.5.15

Version 4.5.15 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.5.15

Bugs Fixed

- Incorrect version for mod_wsgi was being reported in server token.
- On 32 bit platforms, when reading from request content, all input would be returned and the chunk size would be ignored.

Version 4.5.14

Version 4.5.14 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.5.14

Bugs Fixed

- Using the `--url-alias` option to the `runmodwsgi` management command when integrating `mod_wsgi-express` with Django could fail with Python 3. This is because the type of the items passed in an option list could be tuple or list depending on Python version. It was necessary to add items with same type else sorting would break.

New Features

- Added a `name` attribute to the log object used in place of `sys.stdout` and `sys.stderr`, and which is also used for `wsgi.errors` in the per request `environ` dictionary. This is because although the `name` attribute is not required to exist, one can find code out there that assumes it always does exist for file like objects. Adding the attribute ensures that such code doesn't fail.

Version 4.5.13

Version 4.5.13 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.5.13

New Features

- Added `response-socket-timeout` option to `WSGIDaemonProcess` directive to allow the timeout on writes back to HTTP client from Apache child worker process, when proxying responses from a mod_wsgi daemon process, to be separately overridden. Previously this would use the value of the Apache `Timeout` directive. With this change the timeout will be based on `response-socket-timeout` option. If that is not set it will use the the general `socket-timeout` option and if that isn't set only then will the value of the Apache `Timeout` directive be used.

  The overall purpose of being able to separately control this option is to combat against HTTP clients that never read the response, causing the response buffer when proxying to fill up, which in turn can cause the request thread in the daemon process to block. The default high value of the Apache `Timeout` directive, at 300 seconds meant it could take a while to clear, and if the mod_wsgi daemon processes were configured with a low total number of request threads, the whole WSGI application could block if this occurred for many requests at the same time.

  When using `mod_wsgi-express` the option can be set using the command line `--response-socket-timeout` option. If using `mod_wsgi-express` the default socket timeout is 60 seconds so the issue would not have had as big an impact, especially since `mod_wsgi-express` also defines a default request timeout of 60 seconds, which would have resulted in the daemon process being restarted if the request had blocked in returning the response.

  An additional error message is also now logged to indicate that failure to proxy the response content was due to a socket timeout. This will help to indentify where problems are due to a blocked connection or slow client.

Version 4.5.12

Version 4.5.12 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.5.12

Bugs Fixed

- When the `pip install` method is used to compile the module for Windows, the `mod_wsgi-express module-config` command was generating the wrong DLL path for `LoadFile` directive for Python 3.4, as well as possibly older Python versions.

New Features

- When using `pip install` on Windows, in addition to looking in the directory `C:\Apache24` for an Apache installation, it will now also check `C:\Apache22` and `C:\Apache2`. It is recommended though that you use Apache 2.4. If your Apache installation is elsewhere, you can still set the `MOD_WSGI_APACHE_ROOTDIR` environment variable to its location. The environment variable should be set in your shell before running `pip install mod_wsgi` and should be set in a way that exports it to child processes run from the shell.
- Added `restart-interval` option to `WSGIDaemonProcess` for restarting daemon mode processes after a set time. If `graceful-timeout` option is also specified, active requests will be given a chance to complete, while still accepting new requests. If within the grace period the process becomes idle, a shutdown will occur immediately. In the case of no grace period being specified, or the grace period expiring, the normal shutdown sequence will occur. The option is also available in `mod_wsgi-express` as `--restart-interval`.

Version 4.5.11

Version 4.5.11 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.5.11

Bugs Fixed

- The `runmodwsgi` option when using Django application integration would fail on older Django versions up to Django 1.7.

Version 4.5.10

Version 4.5.10 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.5.10

Bugs Fixed

- In version 4.5.9, the version number 4.5.8 was being incorrectly reported via `mod_wsgi.version` in the per request WSGI environ dictionary.
- When using Anaconda Python on MacOS X, the Python shared library wasn't being resolved correctly due to changes in Anaconda Python, meaning it cannot be used in embedded systems which load Python via a dynamically loaded module, such as in Apache. When using `mod_wsgi-express` the Python shared library is now forcibly loaded before the mod_wsgi module is loaded in Apache. If doing manual Apache configration, you will need to add before the `LoadModule` line for `wsgi_module`, a `LoadFile` directive which loads the Ananconda Python shared library by its full path from where it is located in the Anaconda Python `lib` directory.
- Startup timeout wasn't being cancelled after succesful load of the WSGI script file and instead was only being done after first request had finished. This meant that if first request took longer than the startup timeout the process would be wrongly restarted.
- Fix parsing of `Content-Length` header returned in daemon mode so that responses greater than 2GB in size could be returned.
- Using incorrect header files in workaround to be able to compile mod_wsgi on MacOSX Sierra when using `pip install`. Was using old MacOS X 10.6 SDK which are header files for Apache 2.2. Was running, but should not have worked at all. Possibility this still may not work or might break. No choice until Apple fixes their broken Xcode and Apache installation.

Version 4.5.9

Version 4.5.9 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.5.9

Bugs Fixed

- Revert `apachectl` script generated by `mod_wsgi-express` back to using `/bin/bash` as `/bin/sh` on some Linux systems lacking ability to do `exec -a`.

Version 4.5.8

Version 4.5.8 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.5.8

Bugs Fixed

- When using HTTP/2 support and `wsgi.file_wrapper`, the response could be truncated when `mod_h2` was deferring the sending of the response until after the WSGI request had been finalized.
- Builds were failing on Windows. Insert appropriate `#if` conditional around code which shouldn't have been getting included on Windows.
- When `mod_wsgi-express` is run as `root` and `--python-eggs` option is used, if the directory for the Python eggs didn't exist, it was created, but the ownership/group were not set to be the user and group that Apache would run the WSGI application. As a result Python eggs could not actually be unpacked into the directory. Now change the ownership/group of the directory to user/group specified when `mod_wsgi-express` was run.
- Installation on MacOS X Sierra fails for both CMMI and `pip install` methods. This is because Apple removed `apr-1-config` and `apu-1-config` tools needed by `apxs` to install third party Apache module. A workaround has been incorporated so that installation still works when using `pip install`, but there is no workaround for CMMI method. You will need to use `pip install` method and then use `mod_wsgi-express module-config` to get the configuration to then add into the Apache configuration so it knows how to load the mod_wsgi module. Then configure Apache so it knows about your WSGI application.
- Compilation would fail on MacOS X Sierra as the API was changed for obtaining task information. This was used to get memory used by the process.

New Features

- Add `WSGIIgnoreActivity` directive. This can be set to `On` inside of a `Location` directive block for a specific URL path, and any requests against matching URLs will not trigger a reset of the inactivity timeout for a mod_wsgi daemon process. This can be used on health check URLs so that periodic requests against the health check URL do not interfere with the inactivity timeout and keep the process running, rather than allowing the process to restart due to being otherwise idle.
- Added the `--ignore-activity` option to `mod_wsgi-express`. It will set the `WSGIIgnoreActivity` directive to `On` for the specific URL path passed as argument to the option. Any requests against the matching URL path will not trigger a reset of the inactivity timeout for a mod_wsgi daemon process.
- Added the `--module-config` option to `mod_wsgi-express` to get the Apache configuration snippet you would use to load the mod_wsgi module from the Python installation direct into Apache, rather than installing the module into the Apache modules directory.
- Added experimental support for installing mod_wsgi on Windows using `pip`. Is only tested with Apache 2.4 and Python 3.5. The Apache installation must be installed in `C:\Apache24` directory. Run `pip install mod_wsgi`. The run `mod_wsgi-express module-config` and it will generate the required configuration to add into the Apache configuration file to load the mod_wsgi module. You still need to separately configure Apache for your specific WSGI application.

Version 4.5.7

Version 4.5.7 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.5.7

Bugs Fixed

1. Resolved problem whereby mod_wsgi would fail on startup when using Anaconda Python. This was caused by Anaconda Python changing the behaviour of the C API function `Py_GetVersion()` so that it can no longer be called before the Python interpreter is initialised. Now display only the Python major and minor version in server string from time of compilation, rather than runtime. Also no longer log warning about mismatches between compile time and runtime Python version. This avoids need to call `Py_GetVersion()`.

New Features

1. Add `--http2` option to `mod_wsgi-express` for enabling support of HTTP/2. Requires the `mod_http2` module to be compiled into Apache httpd server for versions of Apache where that is available.

Version 4.5.6

Version 4.5.6 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.5.6

Bugs Fixed

1. Reinstanted change to associate any messages logged via `sys.stdout` and `sys.stderr` back to the request so that Apache can log them with the correct request log ID. This change was added in 4.5.4, but was reverted in 4.5.5 as the change was causing process crashes under Python 3.
2. When using Apache 2.4 use new style `Require` directive instead of older `Order` and `Allow` when setting up access controls for `mod_wsgi-express`. This fixes a problem where when using `--include-file` and `Require` directive was being used. Precedence order was such that older directives were overriding new directive and it was possible to permit access to additional directories when using custom configuration.
3. Django 1.10 requires that management commands use argparse style options but `mod_wsgi-express` uses optparse style options. Can no longer simply merge main script option list to get management command option list. Instead need to convert optparse list to argparse format on the fly, as still need to retain main script option list as optparse until drop Python 2.6 support. Changes stop `runmodwsgi` management command failing when using Django 1.10+.

New Features

1. Added `startup-timeout` option to `WSGIDaemonProcess` directive. If set and the first loading of the WSGI application script file fails, then if no subsequent attempt to load it succeeds within the specified startup timeout, the daemon process will be restarted. When configuring mod_wsgi directly, the option is not enabled by default. The option is exposed via `mod_wsgi-express` with a default value of 15 seconds.

   This would be used where running the Django web framework and there is a risk of the database not being available, causing Django initialisation to fail. Django doesn't allow initialisation to be performed a second

time in the same process, meaning it will then constantly fail. Use of startup timeout will allow the process to be restarted in face of such constant startup failures. If the database is available when the process is restarted, then next time the process starts, everything should be fine.

Do note that this option should preferably only be used where the one WSGI application has been delegated to a WSGI daemon process. This is because if multiple WSGI applications are hosted out of the daemon process group, be they in the same application group or distinct ones, as soon as any one of them loads successfully, then the startup timeout is disabled, meaning that if a subsequent one loaded is constantly failing, then a process restart will not occur. Best practice is to delegate each WSGI application to a distinct daemon process group.

Version 4.5.5

Version 4.5.5 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.5.5

Features Changed

1. Reverted the change in 4.5.4 which associated any messages logged via `sys.stdout` and `sys.stderr` back to the request so that Apache could log them with the correct request log ID. This was necessary as the change was causing process crashes under Python 3. The feature will be reinstated when a solution to the issue can be found.

Version 4.5.4

Version 4.5.4 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.5.4

Bugs Fixed

1. When using Apache 2.4 and daemon mode, the connection and request log IDs from the Apache child worker processes were not being copied across to the daemon process so that log messages generated against the request would use the same ID in logs when using the `%L` format modifier.
2. When using Apache 2.4 and daemon mode, the remote client port information was not being cached such that log messages generated against the request would use the port in logs when using the `%a` format modifier.

Features Changed

1. If `sys.stdout` and `sys.stderr` are used in the context of the thread handling a request, calls against them to log messages will be routed back via `wsgi.errors` from the per request WSGI `environ` dictionary. This avoids the danger of logged messages from different request handlers being intermixed as buffering will now be done on a per request basis. Such messages will also be logged with the correct connection and request log ID if the `%L` formatter is used in the error log format.

New Features

1. Added new option `--error-log-format` to `mod_wsgi-express` to allow the error log message format to be specified.
2. Pass through to the WSGI per request `environ` dictionary new values for `mod_wsgi.connection_id` and `mod_wsgi.request_id`. These are the Apache log IDs for the connection and request that it uses in log messages when using the `%L` format modifier. This only applies to Apache 2.4 and later.

Version 4.5.3

Version 4.5.3 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.5.3

Bugs Fixed

1. Ensure that startup messages are flushed so immediately visible in logs.

Version 4.5.2

Version 4.5.2 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.5.2

Bugs Fixed

1. When using `--debug-mode` with `mod_wsgi-express` any additional directories to search for Python modules, which were supplied by the `--python-path` option, were not being added to `sys.path`.

Version 4.5.1

Version 4.5.1 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.5.1

Bugs Fixed

1. The CPU user and system time for requests wasn't always being output in request finished event data.

Version 4.5.0

Version 4.5.0 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.5.0

New Features

1. Added additional internal performance monitoring features, included per request event mechanism for getting extended metrics on a per request basis. This includes details like per request CPU burn, which along with process level CPU burn and thread utilisation can be used to better tune processes/threads settings.

Version 4.4.23

Version 4.4.23 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.4.23

For details on the availability of Windows binaries see:

https://github.com/GrahamDumpleton/mod_wsgi/tree/master/win32

New Features

1. Added the `--ssl-certificate-chain-file` option to `mod_wsgi-express`, for specifying the path to a file containing the certificates of Certification Authorities (CA) which form the certificate chain of the server certificate. This is equivalent to having used the Apache `SSLCertificateChainFile` directive.

Version 4.4.22

Version 4.4.22 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.4.22

For details on the availability of Windows binaries see:

https://github.com/GrahamDumpleton/mod_wsgi/tree/master/win32

Bugs Fixed

1. Stack traces logged at `INFO` level when a request timeout occurred were not displaying correctly when Python 3 was being used. It is possible that the logging code could also have caused the process to then crash as the process was shutting down.

2. When using the `--url-alias` option with `mod_wsgi-express` and the target directory had a trailing slash, that trailing slash was being incorrectly dropped. This would cause URL lookup to fail when the URL for the directory was a sub URL and also had a trailing slash.

New Features

1. When using `mod_wsgi-express`, rewrite rules can now be added into the `rewrite.conf` file located under the server root directory. An alternate location for the rewrite rules can be specified using the `--rewrite-rules` option.

Note that the rewrite rules are included within a `Directory` block of the Apache configuration file, for the document root directory. Any rules therefore needs to be written so as to work in this context.

If you need to debug the rewrite rules and are using Apache 2.4, the easiest way to enable rewrite logging is to use the `--log-level` option with the quoted value of `'info rewrite:trace8'`.

Version 4.4.21

Version 4.4.21 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.4.21

For details on the availability of Windows binaries see:

https://github.com/GrahamDumpleton/mod_wsgi/tree/master/win32

Features Changed

1. When any of the options `--enable-debugger`, `--enable-debugger`, `--enable-coverage`, `--enable-profiler`, `--enable-recorder` or `--enable-gdb` are used, debug module will now automatically be enabled. Previously you had to also supply the `--debug-mode` option otherwise these options wouldn't be honoured.

New Features

1. Add a WSGI test application to `mod_wsgi-express` which returns back details of the request headers, application environment and request content as the response. This can be used for testing how requests are passed through and also what the execution environment looks like. It can be used by running:

```
mod_wsgi-express start-server --application-type module mod_wsgi.server.environ
```

2. Added `--entry-point` option to `mod_wsgi-express` as more explicit way of identifying the file or module name containing the WSGI application entry point or description. This is in addition to simply being able to list it without any option. The explicit way just makes it easier to see the purpose when you have a long list of options.

Version 4.4.20

Version 4.4.20 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.4.20

For details on the availability of Windows binaries see:

https://github.com/GrahamDumpleton/mod_wsgi/tree/master/win32

Bugs Fixed

1. Post mortem debugger would fail if the exception was raised during yielding of items from a WSGI application, or inside of any `close()` callable of an iterator returned from the WSGI application.

Version 4.4.19

Version 4.4.19 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.4.19

For details on the availability of Windows binaries see:

https://github.com/GrahamDumpleton/mod_wsgi/tree/master/win32

Bugs Fixed

1. Daemon mode processes were crashing when attempting to set `USER`, `USERNAME`, `LOGNAME` or `HOME` when no password entry could be found for the current user ID. Now do not attempt to set these if the user ID doesn't have a password file entry.

Version 4.4.18

Version 4.4.18 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.4.18

For details on the availability of Windows binaries see:

https://github.com/GrahamDumpleton/mod_wsgi/tree/master/win32

Bugs Fixed

1. If `mod_wsgi-express` was run under a user ID for which there was no password entry in the system password file, it would fail when looking up the group name. If this occurs now use `#nnn` as the default group name, where `nnn` is the user ID.

Version 4.4.17

Version 4.4.17 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.4.17

For details on the availability of Windows binaries see:

https://github.com/GrahamDumpleton/mod_wsgi/tree/master/win32

Bugs Fixed

1. If `mod_wsgi-express` was run under a user ID for which there was no password entry in the system password file, it would fail when looking up the user name. If this occurs now use `#nnn` as the default user name, where `nnn` is the user ID.

Version 4.4.16

Version 4.4.16 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.4.16

For details on the availability of Windows binaries see:

https://github.com/GrahamDumpleton/mod_wsgi/tree/master/win32

Bugs Fixed

1. If `/dev/stderr` cannot be opened for writing when startup log is requested and logging to the terminal, then `mod_wsgi-express` would fail. Now attempt fallback to using `/dev/tty` and if that cannot be opened either, then give up on trying to use terminal for startup log.

Version 4.4.15

Version 4.4.15 of mod_wsgi can be obtained from:

> https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.4.15

For details on the availability of Windows binaries see:

> https://github.com/GrahamDumpleton/mod_wsgi/tree/master/win32

Bugs Fixed

1. When specifying multiple directories for the Python module search path using the `WSGIPythonPath` directive, or the `python-path` option to `WSGIDaemonProcess`, it was failing under Python 3 due to incorrect logging. It was therefore only possible to add a single directory.

2. If Apache was already running when the mod_wsgi module was enabled or otherwise configured to be loaded, and then an Apache graceful restart was done so that it would be loaded for the first time, all child processes would crash when starting up and would keep crashing, requiring Apache be shutdown. This would occur as Python initialisation was not being performed correctly in this specific case where mod_wsgi was loaded when Apache was already running and a graceful restart, rather than a normal restart was done.

Version 4.4.14

Version 4.4.14 of mod_wsgi can be obtained from:

> https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.4.14

For details on the availability of Windows binaries see:

> https://github.com/GrahamDumpleton/mod_wsgi/tree/master/win32

Bugs Fixed

1. The `--compress-responses` option of `mod_wsgi-express` was failing when Apache 2.4 was used. This was because `mod_filter` module is required when using Apache 2.4 and it wasn't being loaded.

2. On Python 3, the IO object wrapped by `sys.stdout` and `sys.stderr`, according to the Python documentation, must provide a `fileno()` method even though no file descriptor exists corresponding to the Apache error logs. The method should raise `IOError` if called to indicate not file descriptor can be returned.

Previously, an attempt to use `fileno()` on `sys.stdout` and `sys.stderr` would raise an `AttributeError` instead due to there being no `fileno()` method.

3. Use compiler include flags from running of `apr-config` and `apu-config` when doing `pip` install of `mod_wsgi-express`. This is necessary as on MacOS X 10.11 El Capitan the include flags for APR returned by `apxs` refer to the wrong location causing installation to fail.

New Features

1. When proxying a URL path or a virtual host, now setting request header for `X-Forwarded-Port` so back end knows correct port that front end used.

2. When proxying a URL path, if the request came in over a secure HTTP connection, now setting request header for `X-Forwarded-Scheme` so back end knows that front end handled the request over a secure connection. The value of the header will be `https`.

3. When using `mod_wsgi-express`, it is now possible to supply the `--with-cgi` option, with any files in the document root directory with a '.cgi' extension then being processed as traditional CGI scripts.

Version 4.4.13

Version 4.4.13 of mod_wsgi can be obtained from:

> https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.4.13

For details on the availability of Windows binaries see:

> https://github.com/GrahamDumpleton/mod_wsgi/tree/master/win32

Bugs Fixed

1. The pip installable 'mod_wsgi' package was failing to install on OpenShift and Heroku as `mod_wsgi-apxs` isn't used for tarball based installs.

Features Changed

1. For `mod_wsgi-express`, only the web server type is now shown in the server tokens sent back in the `Server` response header. This prevents users from knowing any specifics and thus using that to determine possible vulnerabilities.

New Features

1. Set environment variables from `apachectl` for `mod_wsgi-express` about the server environment which can be used in additional Apache configuration included into the generated configuration. The environment variables are:

- *MOD_WSGI_SERVER_ROOT* - This is the directory where the generated configuration files, startup scripts, etc were placed.
- *MOD_WSGI_WORKING_DIRECTORY* - This is the directory which will be used as the current working directory of the process. Would default to being the same as `MOD_WSGI_SERVER_ROOT` if not overridden.
- *MOD_WSGI_LISTENER_HOST* - The host name or IP on which connections are being accepted. This should only be used if the Apache configuration variable `MOD_WSGI_WITH_LISTENER_HOST` is defined.
- *MOD_WSGI_HTTP_PORT* - The port on which HTTP connections are being accepted.
- *MOD_WSGI_HTTPS_PORT* - The port on which HTTPS connections are being accepted. This should only be used if the Apache configuration variable `MOD_WSGI_WITH_HTTPS` is defined.
- *MOD_WSGI_MODULES_DIRECTORY* - The directory where the Apache modules are installed.
- *MOD_WSGI_RUN_USER* - The user that the WSGI application will be run as.

- *MOD_WSGI_RUN_GROUP* - The group that the WSGI application will be run as.

2. Added `X-Client-IP` to list of possible trusted headers indicating the true remote address of client when passing through a proxy.

Version 4.4.12

Version 4.4.12 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.4.12

For details on the availability of Windows binaries see:

https://github.com/GrahamDumpleton/mod_wsgi/tree/master/win32

Bugs Fixed

1. If the WSGI application when run under daemon mode returned response content as many small blocks, this could result in excessive memory usage in the Apache child worker process proxying the request due to many buckets being buffered until the buffer size threshold was reached. If the number of buckets reaches a builtin threshold the buffered data will now be forcibly flushed even if the size threshold hadn't been reached.

Version 4.4.11

Version 4.4.11 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.4.11

For details on the availability of Windows binaries see:

https://github.com/GrahamDumpleton/mod_wsgi/tree/master/win32

Bugs Fixed

1. No provision was made for operating systems with a very low limit on the number of separate data blocks that could be passed to system `writev()` call. This was an issue on Solaris where the limit is 16 and meant that since version 4.4.0, daemon mode of mod_wsgi would fail where a HTTP request had more than a small number of headers.

2. When installing the `mod_wsgi` package using `pip` and rather than activating the virtual environment you were referring to `pip` by path from the `bin` directory, the `mod_wsgi-httpd` package which had already been installed into the virtual environment would not be detected.

New Features

1. Added the `--service-log` option to `mod_wsgi-express` for specifying the name of a log file for a specific service script. The arguments are the name of the service and the file name for the log. The log file will be placed in the log directory, be it the default, or a specific log directory if specified.

2. Set various environment variables from `mod_wsgi-express` to identify that it is being used, what hosts it is handling requests for, and whether debug mode and/or specific debug mode features are enabled. This is so that a web application can modify it's behaviour when `mod_wsgi-express` is being used, or being used in specific ways. The environment variables which are set are:

- *MOD_WSGI_EXPRESS* - Indicates that `mod_wsgi-express` is being used.
- *MOD_WSGI_SERVER_NAME* - The primary server host name for the site.
- *MOD_WSGI_SERVER_ALIASES* - Secondary host names the site is known by.
- *MOD_WSGI_RELOADER_ENABLED* - Indicates if source code reloading enabled.
- *MOD_WSGI_DEBUG_MODE* - Indicates if debug mode has been enabled.
- *MOD_WSGI_DEBUGGER_ENABLED* - Indicates pdb debugger has been enabled.
- *MOD_WSGI_COVERAGE_ENABLED* - Indicates if coverage analysis has been enabled.
- *MOD_WSGI_PROFILER_ENABLED* - Indicates if code profiling has been enabled.
- *MOD_WSGI_RECORDER_ENABLED* - Indicates if request/response recording enabled.
- *MOD_WSGI_GDB_ENABLED* - Indicates if gdb process crash debugging enabled.

For any environment variable indicating a feature has been enabled, it will be set when enabled and have the value 'true'.

For the list of server aliases, it will be a space separated list of host names.

Version 4.4.10

Version 4.4.10 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.4.10

For details on the availability of Windows binaries see:

https://github.com/GrahamDumpleton/mod_wsgi/tree/master/win32

Bugs Fixed

1. Fixed a reference counting bug which would cause a daemon process to crash if both `home` and `python-path` options were specified at the same time with the `WSGIDaemonProcess` directive.

2. When using `--https-only` option with `mod_wsgi-express`, the redirection from the `http` address to the `https` address was not setting the correct port for `https`.

Features Changed

1. Changed the default Apache log level for `mod_wsgi-express` to `warn` instead of `info`. This has been done avoid very noisy logs when enabling secure HTTP connections. To set back to `info` level use the `--log-level` option.

2. When specifying a service script with the `--service-script` option of `mod_wsgi-express`, the home directory for the process will now be set to the same home directory as used for the hosted WGSI application. Python modules from the WSGI application will therefore be automatically found when imported. Any directory paths added using `--python-path` option will also be added as search directories for Python module imports, with any `.pth` files in those directories also being handled. In addition, the language locale and Python eggs directory used by the hosted WSGI application will also be used for the service script.

3. When specifying `--python-path` option, when paths are now setup for the WSGI application, they will be added in such a way that they appear at the head of `sys.path` and any `.pth` files in those directories are also handled.

New Features

1. Added the `--directory-listing` option to `mod_wsgi-express` to allow automatic directory listings to be enabled when using the static file application type and no explicit directory index file has been specified.

2. In addition to the convenience function of `--ssl-certificate` for `mod_wsgi-express`, which allowed the SSL certificate and private key file to be specified using one option by specifying the command file name up to the extension, separate `--ssl-certificate-file` and `--ssl-certificate-key-file` options are now also provided. These would either both need to be specified, or the existing `--ssl-certificate` option used, when specifying that secure HTTPS connections should be used through having specified `--https-port`.

3. Added the `--ssl-ca-certificate-file` option to `mod_wsgi-express`. If specified this should give the location of the file with any CA certificates to be used for client authentication. As soon as this option is provided, the client authentication will be required for the whole site. This would generally be used in conjunction with the `--https-only` option so that only a secure communication channel is being used.

If you do not wish for the whole site to required client authentication, you can use the `--ssl-verify-client` option to specify sub URLs for which client authentication should be performed.

4. Added the `--ssl-environment` option to `mod_wsgi-express` to enable the passing of standard SSL variables in the WSGI environ dictionary passed to the WSGI application.

5. Added the `WSGITrustedProxies` directive and corresponding option of `--trust-proxy` to `mod_wsgi-express`. This works in conjunction with the `WSGITrustedProxyHeaders` directive and `--trust-proxy-header` option of `mod_wsgi-express`. When trusted proxies are specified, then proxy headers will only be trusted if the request originated with a trusted proxy. Further, any IP addresses corresponding to a proxy listed in the `X-Forwarded-For` header will only be trusted if specified. When determining the value for `REMOTE_ADDR` the IP preceding the last recognised proxy the request passed through will be used and not simply the first IP listed in the header. The header will be rewritten to reflect what was honoured with client IPs of dubious origin discarded.

Version 4.4.9

Version 4.4.9 of mod_wsgi can be obtained from:

> https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.4.9

For details on the availability of Windows binaries see:

> https://github.com/GrahamDumpleton/mod_wsgi/tree/master/win32

Features Changed

1. The `--proxy-url-alias` option of `mod_wsgi-express` has been superseded by the `--proxy-mount-point` option. This option now should only be used to proxy to a whole site or sub site and not individual file resources. If the mount point URL for what should be proxied doesn't have a trailing slash, the trailing slash redirection will first be performed on the proxy for the mount point rather than simply passing it through to the backend.

2. The signal handler intercept will now be removed automatically from a Python child process forked from either an Apache child process or a daemon process. This avoids the requirement of setting `WSGIRestrictSignal` to `Off` if wanting to setup new signal handlers from a forked child process.

3. The signal handler registrations setup in daemon processes to manage process shutdown, will now revert to exiting the process when invoked from a Python process forked from a daemon process. This avoids the need to set new signal handlers in forked processes to override what was inherited.

Note that this only applies to processes forked from daemon mode processes. If you are forking processes when your WSGI application is running in embedded mode, it is still a good idea to set signal handles for `SIGINT`, `SIGTERM` and `SIGUSR1` back to `SIG_DFL` using `signal.signal()` if you want to avoid the possibility of strange behaviour due to the inherited Apache child worker process signal registrations.

New Features

1. Added `--hsts-policy` option to `mod_wsgi-express` to allow a HSTS (`Strict-Transport-Security`) policy response header to be specified which should be included when the `--https-only` option is used to ensure that the site only accepts HTTPS connections.

2. Added `WSGITrustedProxyHeaders` directive. This allows you to specify a space separated list of inbound HTTP headers used to transfer client connection information from a proxy to a backend server, that are trusted. When the specified headers are seen in a request, the values passed via them will be used to fix up the values in the WSGI `environ` dictionary to reflect client information as was seen by the proxy.

Only the specific headers you are expecting and which is guaranteed to have only been set by the proxy should be listed. Whether it exists or not, all other headers in a category will be removed so as to avoid an issue with a forged header getting through to a WSGI middleware which is looking for a different header and subsequently overriding whatever the trusted header specified. This applies to the following as well when more than one convention is used for the header name.

The header names which are accepted for specifying the HTTP scheme used are `X-Forwarded-Proto`, `X-Forwarded-Scheme` and `X-Scheme`. It is expected that the value these supply will be `http` or `https`. When it is `https`, the `wsgi.url_scheme` value in the WSGI `environ` dictionary will be overridden to be `https`.

Alternate headers accepted are `X-Forwarded-HTTPS`, `X-Forwarded-SSL` and `X-HTTPS`. If these are passed, the value needs to be `On`, `true` or `1`. A case insensitive match is performed. When matched, the `wsgi.url_scheme` value in the WSGI `environ` dictionary will be overridden to be `https`.

The header names which are accepted for specifying the target host are `X-Forwarded-Host` and `X-Host`. When found, the value will be used to override the `HTTP_HOST` value in the WSGI `environ` dictionary.

The sole header name accepted for specifying the front end proxy server name is `X-Fowarded-Server`. When found, the value will be used to override the `SERVER_NAME` value in the WSGI `environ` dictionary.

The sole header name accepted for specifying the front end proxy server port is `X-Fowarded-Port`. When found, the value will be used to override the `SERVER_PORT` value in the WSGI `environ` dictionary.

The header names accepted for specifying the client IP address are `X-Forwarded-For` and `X-Real-IP`. When `X-Forwarded-For` is used then the first IP address listed in the header value will be used. For `X-Real-IP` only one IP address should be given. When found, the value will be used to override the `REMOTE_ADDR` value in the WSGI `environ` dictionary.

Note that at present there is no facility for specifying a list of trusted IP addresses to be specified for front end proxies. This will be a feature added in a future version. When that is available and `X-Forwarded-For` is used, then the IP address preceding the furthest away trusted proxy IP address will instead be used, even if not the first in the list.

The header names accepted for specifying the application mount point are `X-Script-Name` and `X-Forwarded-Script-Name`. When found, the value will override the `SCRIPT_NAME` value in the `WSGI` environ dictionary.

When using `mod_wsgi-express` the equivalent command line option is `--trust-proxy-header`. The option can be used multiple times to specify more than one header.

Version 4.4.8

Version 4.4.8 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.4.8

For details on the availability of Windows binaries see:

https://github.com/GrahamDumpleton/mod_wsgi/tree/master/win32

Bugs Fixed

1. The eviction timeout was not being correctly applied when request timeout wasn't being applied at the same time. It may have partly worked if any of inactivity or graceful timeout were also specified, but the application of the timeout may still have been delayed.

New Features

1. Added the `--error-log-name` option to `mod_wsgi-express` to allow the name of the file used for the error log, when being written to the log directory, to be overridden.

2. Added the `--access-log-name` option to `mod_wsgi-express` to allow the name of the file used for the access log, when being written to the log directory, to be overridden.

3. Added the `--startup-log-name` option to `mod_wsgi-express` to allow the name of the file used for the startup log, when being written to the log directory, to be overridden.

Version 4.4.7

Version 4.4.7 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.4.7

For details on the availability of Windows binaries see:

https://github.com/GrahamDumpleton/mod_wsgi/tree/master/win32

Features Changed

1. The `proxy-buffer-size` option to `WSGIDaemonProcess` directive was renamed to `response-buffer-size` to avoid confusion with options related to normal HTTP proxying. The `--proxy-buffer-size` option of `mod_wsgi-express` was similarly renamed to `--response-buffer-size`.

New Features

1. Added `--service-script` option to `mod_wsgi-express` to allow a Python script to be loaded and executed in the context of a distinct daemon process. This can be used for executing a service to be managed by Apache, even though it is a distinct application. The options take two arguments, a short name for the service and the path to the Python script for starting the service.

If `mod_wsgi-express` is being run as root, then a user and group can be specified for the service using the `--service-user` and `--service-group` options. The options take two arguments, a short name for the service and the user or group name respectively.

2. Added `--proxy-url-alias` option to `mod_wsgi-express` for setting up proxying of a sub URL of the site to a remote URL.

3. Added `--proxy-virtual-host` option to `mod_wsgi-express` for setting up proxying of a whole virtual host to a remote URL. Only supports proxying of HTTP requests and not HTTPS requests.

4. Added `eviction-timeout` option to `WSGIDaemonProcess` directive. For the case where the graceful restart signal, usually `SIGUSR1`, is sent to a daemon process to evict the WSGI application and restart the process, this controls how many seconds the process will wait, while still accepting new requests, before it reaches an idle state with no active requests and shuts down.

The `graceful-timeout` option previously performed this exact role in this case previously, but a separate option is being added to allow a different timeout period to be specified for the case for forced eviction. The existing `graceful-timeout` option is still used when a maximum requests option or CPU usage limit is set. For backwards compatibility, if `eviction-timeout` isn't set, it will fall back to using any value specified using the `graceful-timeout` option.

The `--eviction-timeout` option has also been added to `mod_wsgi-express` and behaves in a similar fashion.

5. Added support for new `mod_wsgi-httpd` package. The `mod_wsgi-httpd` package is a pip installable package which will build the Apache httpd server and install it into the Python installation. If the `mod_wsgi-httpd` package is installed before installing this package, then the Apache httpd server installation installed by `mod_wsgi-httpd` will be used instead of any system installed version of the Apache httpd server when running `mod_wsgi-express`. This allows you to workaround any inability to upgrade the main Apache installation, or install its 'dev' package if missing, or install it outright if not present.

Version 4.4.6

Version 4.4.6 of mod_wsgi can be obtained from:

> https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.4.6

For details on the availability of Windows binaries see:

> https://github.com/GrahamDumpleton/mod_wsgi/tree/master/win32

Bugs Fixed

1. Apache 2.2.29 and 2.4.11 introduce additional fields to the request structure `request_rec` due to CVE-2013-5704. The addition of these fields will cause versions of mod_wsgi from 4.4.0-4.4.5 to crash when used in mod_wsgi daemon mode and mod_wsgi isn't initialising the new structure members.

If you are upgrading your Apache installation to those versions or later versions, you must also update to mod_wsgi version 4.4.6. The mod_wsgi 4.4.6 source code must have also been compiled against the newer Apache version.

In recompiling mod_wsgi 4.4.6 source code against the newer Apache versions the source code is able to detect the new fields exist at compile time by checking a compile time version number.

One problem that can arise is that where a CVE is raised for a security issue, Linux distributions will back port the change to older Apache versions. When they do this though, the compile time version number isn't changed, so mod_wsgi cannot detect at compile time when built against Apache versions with the backport that the additional fields exist.

To combat this problem, mod_wsgi will do some runtime checks which look at the actual size of `request_rec` and calculate whether the additional fields have been added by way of a backported change. In this case mod_wsgi will then set the fields as necessary.

As a final fail safe for forward compatibility. If the current mod_wsgi source code is compiled against a version of Apache which doesn't have the CVE change applied, it will pad the `request_rec` and optimistically set the fields anyway. This is to deal with the situation where mod_wsgi is compiled against an older Apache and then that Apache is upgraded to one with the CVE change, but mod_wsgi is not recompiled so that the additional fields can be detected at compile time.

2. Override `LC_ALL` environment variable when `locale` option to the `WSGIDaemonProcess` directive. It is not always sufficient to just call `setlocale()` as some Python code, including interpreter initialisation can still consult the original `LC_ALL` environment variable. In this case this can result in an undesired file system encoding still being selected.

New Features

1. Added `--enable-gdb` option to `mod_wsgi-express` for when running in debug mode. With this option set, Apache will be started up within `gdb` allowing the debug of process crashes on startup or while handling requests. If the `gdb` program is not in `PATH`, the `--gdb-executable` option can be set to give its location.

Version 4.4.5

Version 4.4.5 of mod_wsgi can be obtained from:

> https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.4.5

Known Issues

1. Although the makefiles for building mod_wsgi on Windows have now been updated for the new source code layout, some issues are being seen with mod_wsgi on Apache 2.4. These issues are still being investigated. As most new changes in 4.X relate to mod_wsgi daemon mode, which is not supported under Windows, you should keep using the last available binary for version 3.X on Windows instead. Binaries compiled by a third party can be obtained from:

- http://www.lfd.uci.edu/~gohlke/pythonlibs/#mod_wsgi

Bugs Fixed

1. When installing `mod_wsgi-express` from PyPi on OpenShift as a dependency of an application `setup.py` file, the precompiled Apache binaries would not be installed.

Version 4.4.4

Version 4.4.4 of mod_wsgi can be obtained from:

> https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.4.4

Known Issues

1. Although the makefiles for building mod_wsgi on Windows have now been updated for the new source code layout, some issues are being seen with mod_wsgi on Apache 2.4. These issues are still being investigated. As most new changes in 4.X relate to mod_wsgi daemon mode, which is not supported under Windows, you should keep using the last available binary for version 3.X on Windows instead. Binaries compiled by a third party can be obtained from:

- http://www.lfd.uci.edu/~gohlke/pythonlibs/#mod_wsgi

New Features

1. The `mod_wsgi-express` command will now output to `stdout` the number of daemon processes and threads being used.

2. Add automatic installation of precompiled Apache binaries when deploying `mod_wsgi-express` to Heroku or OpenShift. These binaries will be pulled down from S3 and installed as part of the mod_wsgi package.

Version 4.4.3

Version 4.4.3 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.4.3

Known Issues

1. Although the makefiles for building mod_wsgi on Windows have now been updated for the new source code layout, some issues are being seen with mod_wsgi on Apache 2.4. These issues are still being investigated. As most new changes in 4.X relate to mod_wsgi daemon mode, which is not supported under Windows, you should keep using the last available binary for version 3.X on Windows instead. Binaries compiled by a third party can be obtained from:

- http://www.lfd.uci.edu/~gohlke/pythonlibs/#mod_wsgi

Features Changed

1. The `--lang` option to `mod_wsgi-express` has been deprecated. Any default language locale setting should be set exclusively using the `--locale` option.

2. The behaviour of the `--locale` option to `mod_wsgi-express` has changed. Previously if this option was not defined, then both of the locales `en_US.UTF-8` and `C.UTF-8` have at times been hardwired as the default locale. These locales are though not always present. As a consequence, a new algorithm is now used.

If the `--locale` option is supplied, the argument will be used as the locale. If no argument is supplied, the default locale for the executing `mod_wsgi-express` process will be used. If that however is `C` or `POSIX`, then an attempt will be made to use either the `en_US.UTF-8` or `C.UTF-8` locales and if that is not possible only then fallback to the default locale of the `mod_wsgi-express` process.

In other words, unless you override the default language locale, an attempt is made to use an English language locale with `UTF-8` encoding.

3. Unless the process name is overridden using `--process-name` option to `mod_wsgi-express`, the Apache parent and child worker process will be given a name such as `httpd (mod_wsgi-express)` making them more easily distinguishable from a traditional Apache installation.

Bugs Fixed

1. The `mod_wsgi-express` script would fail on startup if the user had a corresponding group ID which didn't actually match an existing group in the groups file and no override group was being specified. When this occurs, the group will now be specified as `#nnn` where `nnn` is the group ID.

New Features

1. Added `--process-name` option to `mod_wsgi-express` to allow the name of the Apache parent process to be overridden as it would be displayed in `ps`. This is necessary under some process manager systems where it looks for a certain name, but with shell script wrappers and exec calls happening around `mod_wsgi-express` the name would change.

Version 4.4.2

Version 4.4.2 of mod_wsgi can be obtained from:

> https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.4.2

Known Issues

1. Although the makefiles for building mod_wsgi on Windows have now been updated for the new source code layout, some issues are being seen with mod_wsgi on Apache 2.4. These issues are still being investigated. As most new changes in 4.X relate to mod_wsgi daemon mode, which is not supported under Windows, you should keep using the last available binary for version 3.X on Windows instead. Binaries compiled by a third party can be obtained from:

- http://www.lfd.uci.edu/~gohlke/pythonlibs/#mod_wsgi

Features Changed

1. The `--ssl-port` option has been deprecated in favour of the option `--https-port`. Strictly speaking SSL no longer exists and has been supplanted with TLS. The 'S' in 'HTTPS' is actually meant to mean secure and not 'SSL'. So change name of option to properly match terminoligy.

2. The name of the startup log was changed such that naming was consistent with how logs are normally named with Apache. That is `startup_log` instead of `startup.log`, thereby matching convention with `error_log` and `access_log`.

Bugs Fixed

1. When a default language was specified using the `locale` option to the `WSGIDaemonProcess` directive or the `--locale` option to `mod_wsgi-express`, if it did not actually match a locale supported by the operating system, that the locale couldn't be set wasn't logged. Such a message is now logged along with a suggestion to use `C.UTF-8` as a fallback locale if the intent is to have `UTF-8` support.

2. When using the `--https-only` option with `mod_wsgi-express`, a HTTP request was not being redirected to be a HTTPS request when there were no server aliases specified.

Version 4.4.1

Version 4.4.1 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.4.1

Known Issues

1. The makefiles for building mod_wsgi on Windows are currently broken and need updating. As most new changes relate to mod_wsgi daemon mode, which is not supported under Windows, you should keep using the last available binary for version 3.X on Windows instead.

Bugs Fixed

1. Process crashes could occur when request content had been consumed by the WSGI application. The trigger was when the Python `wsgi.input` was still in existence after the web request had finished. The destruction of the `wsgi.input` object was accessing memory which had already been released back to the Apache memory pools and potentially reused. This could cause crashes or other unexplained behaviour. This issue was introduced in version 4.4.0 of mod_wsgi.

Features Changed

1. When an error occurs in writing back a response to the HTTP client, during the consumption of the iterable returned by the WSGI application, the message will now be logged at debug level rather than error level. Note that under Apache 2.2 it isn't possible to suppress the message generated by Apache itself from the core_output_filter, so that may still appear.

2. The `--profiler-output-file` option for `mod_wsgi-express` was changed to `--profiler-directory` and now refers to a directory, with individual pstats files being added to the directory for each session rather than reusing the same name all the time.

New Features

1. Added the `--server-mpm` option to `mod_wsgi-express`. With this option, if you are using Apache 2.4 with dynamically loadable MPM modules and more than one option for the MPM is available, you can specify your preference for which is used. If not specified, then the precedence order for MPMs is 'event', 'worker' and finally 'prefork'.

2. Added `static` as an option for `--application-type` when running `mod_wsgi-express`. When set as `static`, only static files will be served. One can still set specific handler types for different extensions which may invoke a Python handler script, but there will be no global fallback WSGI application for any URLs that do not map to static files. In these cases a normal HTTP 404 response will be returned instead.

3. Added `--host-access-script` option to `mod_wsgi-express` to allow a Python script to be provided which can control host access. This uses the `WSGIAccessScript` directive and the handler script should define an `allow_access(environ, host)` function which returns `True` if access is allowed or `False` if blocked.

4. Added `--debugger-startup` option to be used in conjunction with the `--enable-debugger` option of `mod_wsgi-express` when in debug mode. The option will cause the debugger to be activated on server start before any requests are handled to allow breakpoints to be set.

5. Added a `socket-user` option to `WSGIDaemonProcess` to allow the owner of the UNIX listener socket for the daemon process group to be overridden. This can be used when using mod_ruid2 to change the owner of the

socket from the default Apache user, to the user under which mod_ruid2 will run Apache when handling requests. This is necessary otherwise the Apache child worker process will not be able to connect to the listener socket for the mod_wsgi daemon process to proxy the request to the WSGI application.

6. Added a `--enable-recorder` option for enabling request recording when also using debug mode.

Version 4.4.0

Version 4.4.0 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.4.0

Known Issues

1. The makefiles for building mod_wsgi on Windows are currently broken and need updating. As most new changes relate to mod_wsgi daemon mode, which is not supported under Windows, you should keep using the last available binary for version 3.X on Windows instead.

Bugs Fixed

1. When an exception occurs during the yielding of data from a generator returned from the WSGI application, and chunked transfer encoding was used on the response, then a '0' chunk would be errornously added at the end of the response content even though the response was likely incomplete. The result would be that clents wouldn't be able to properly detect that the response was truncated due to an error. This issue is now fixed for when embedded mode is being used. Fixing it for daemon mode is a bit trickier.

2. Response headers returned from the WSGI application running in daemon mode were being wrongly attached to the internal Apache data structure for `err_headers_out` instead of `headers_out`. This meant that the `Header` directive of the `mod_headers` module, with its default condition of only checking `onsuccess` headers would not work as expected.

In order to be able to check for or modify the response headers one would have had to use the `Header` directive with the `always` condition and if also working with an embedded WSGI application, also define a parallel `Header` directive but with the `onsuccess` condition.

For daemon mode, response headers will now be correctly associated with `headers_out` and the `onsuccess` condition of the `Header` directive. The only exception to this in either embedded or daemon mode now is that of the `WWW-Authenticate` header, which remains associated with `err_headers_out` so that the header will survive an internal redirect such as to an `ErrorDocument`.

3. When optional support for chunked requests was enabled, it was only working properly for embedded mode. The feature now also works properly for daemon mode.

The directive to enable support for chunked request content is `WSGIChunkedRequest`. The command line option when using mod_wsgi express is `--chunked-request`.

This is an optional feature, as the WSGI specification is arguably broken in not catering properly for mutating input filters or chunked request content. Support for chunked request content could be enabled by default, but then WSGI applications which don't simply read all available content and instead rely entirely on `CONTENT_LENGTH`, would likely see a chunked request as having no content at all, as it would interpret the lack of the `CONTENT_LENGTH` as meaning the length of the content is zero.

An attempt to get the WSGI specification ammended to be more sensible and allow what is a growing requirement to support chunked request content was ignored. Thus support is optional. You will need to enable this if you wish to rely on features of any WSGI framework that take the more sensible approach of ignoring `CONTENT_LENGTH` as a true

indicator of content length. One such WSGI framework which provides some support for chunked request content is Flask/Werkzeug. Check its documentation or the code for Flask/Werkzeug to to see if any additional `SetEnv` directive may be required to enable the support in Flask/Werkzeug.

4. Fixed a potential request content data corruption issue when running a WSGI application in daemon mode. The bug in the code is quite obvious, yet unable to trigger it on older mod_wsgi versions. It was though triggering quite easily in the current release on MacOS X, prior to it being fixed, due to the changes made to support chunked request content for daemon processes.

Suspect it is still a latent bug in older mod_wsgi versions, but the conditions under which it would trigger must have been harder to induce. The lack of reported problems may have been aided by virtue of Linux UNIX socket buffer size being quite large, in comparison to MacOS X, and so harder to create a condition where not all data could be written onto the UNIX socket in one call. Yet, when buffer sizes for the UNIX socket on MacOS X were increased, it was still possible to induce the bug.

5. When the `--working-directory` option for `mod_wsgi-express` was given a relative path name, that wasn't being translated to an absolute path name when substituting the `home` option of `WSGIDaemonProcess` causing server startup to fail.

6. When using `--debug-mode` of `mod_wsgi-express` the working directory for the application was not being added to `sys.path`. This meant that if the WSGI script was referenced from a different directory, any module imports for other modules in that directory would fail.

Features Changed

1. Until recently, a failed attempt to change the working directory for a daemon process to the user the process runs as would be ignored. Now it will cause a hard failure that will prevent the daemon process from starting. This would cause issues where the user, usually the default Apache user, has not valid home directory. Now what will happens is that any attempt will only be made to change the working directory to the home directory of the user the daemon process runs as, if the 'user' option had been explicitly set to define the user and the user is different to the user that Apache child worker processes run as. In other words, is different to the default Apache user.

2. The support for the `wdb` debugger was removed. Decided that it wasn't mainstream enough and not ideal that still required a separate service and port to handle debugging sessions.

New Features

1. Added new feature to `mod_wsgi-express` implementing timeouts on the reading of the request, including headers, and the request body. This feature uses the Apache module `mod_reqtimeout` to implement the feature.

By default a read timeout on the initial request including headers of 15 seconds is used. This can dynamically increase up to a maximum of 30 seconds if the request data is received at a minimum required rate.

By default a read timeout on the request body of 15 seconds is used. This can dynamically increase if the request data is received at a minimum required rate.

The options to override the defaults are `--header-timeout`, `--header-max-timeout`, `--header-min-rate`, `--body-timeout`, `--body-max-timeout` and `--body-min-rate`. For a more detailed explaination of this feature, consult the documentation for the Apache `mod_reqtimeout` module.

2. Added a new `%{HOST}` label that can be used when specifying the application group (Python sub interpreter context) to run the WSGI application in, via the `WSGIApplicationGroup` directive, or the `application-group` option to `WSGIScriptAlias`.

This new label will result in an application group being used with a name that corresponds to the name of the site as identified by the HTTP request `Host` header. Where the accepting port number is other than 80 or 443, then the name of the application group will be suffixed with the port number separated by a colon.

Note that extreme care must be exercised when using this new label to specify the application group. This is because the HTTP request `Host` header is under the control of the user of the site.

As such, it should only be used in conjunction with a configuration which adequately blocks access to anything but the expected hosts.

For example, it would be dangerous to use this inside of a `VirtualHost` where the `ServerAlias` directive is used with a wildcard. This is because a user could pick arbitrary host names matching the wildcard and so force a new sub interpreter context to be created each time and so blow out memory usage.

Similarly, caution should be exercised with `mod_vhost_alias`, with any configuration forbidding any host which doesn't specifically match some specified resource such as a directory.

Finally, this should probably never be used when not using either `VirtualHost` or `mod_vhost_alias` as in that case the server is likely going to accept any `Host` header value without exclusions.

3. Allow `%{RESOURCE}`, `%{SERVER}` and `%{HOST}` labels to be used with the `WSGIProcessGroup` directive, or the `process-group` option of the `WSGIScriptAlias` directive.

For this to work, it is still necessary to have setup an appropriate mod_wsgi daemon process group using the `WSGIDaemonProcess` directive, with name that will match the expanded value for the respective labels. If there is no matching mod_wsgi daemon process group specified, then a generic HTTP 500 internal server error response would be returned and the reason, lack of matching mod_wsgi daemon process group, being logged in the Apache error log.

4. Error messages and exceptions raised when there is a failure to read request content, or write back a response now provide the internal error indication from Apache as to why. For the `IOError` exceptions which are raised, that the exception originates within Apache/mod_wsgi is now flagged in the description associated with the exception.

5. When using mod_wsgi daemon mode and there is a timeout when reading request content in order to proxy it to the daemon process, a 408 request timeout HTTP response is now returned where as previously a generic 500 internal server error HTTP response was returned.

Note that this doesn't mean that the WSGI application wasn't actually run. The WSGI application in the daemon process would have run as soon as the headers had been received.

If the WSGI application had actually attempted to read the request content, it should also have eventually received an exception of type `IOError` when accessing `wsgi.input` to read the request content, due to a timeout or due to the proxy connection being closed before all request content was able to be read.

If the WSGI application wasn't expecting any request content and had ignored it, even though some was present, it would still have run to completion and generated a response, but because the Apache child worker process was blocked waiting for content, when the timeout occurred the client would get the 408 HTTP response rather than the actual response generated by the WSGI application.

6. Added the `--log-to-terminal` option to `mod_wsgi-express` to allow the error log output to be directed to standard error for the controlling terminal, and the access log output, if enabled, to be directed to standard output. Similarly, the startup log output, if enabled, will be sent to standard error also.

This should not be used in conjunction with `--setup-only` option when using the generated `apachectl` script, unless the `-DFOREGROUND` option is also being supplied to `apachectl` at the time it is run with the `start` command.

7. Added the `--access-log-format` option to `mod_wsgi-express`. By default if the access log is enabled, entries will follow the 'common' log format as typically used by Apache. You have two options of how you can use the `--access-log-format`. The first is to give it the argument 'combined', which will then cause it to use this alternate log format which is again often used with Apache. The other is to specify the log format string yourself.

The format string can contain format string components as would be used with the `LogFormat` directive. For example, to specify the equivalent to the 'common' log format, you could use:

```
--access-log-format "%h %l %u %t \"%r\" %>s %b"
```

This 'common' log format is identified via a nickname in the same way 'combined' is, so if you did have to specify it explicitly for some reason, you could just have instead used:

```
--access-log-format common
```

8. Added the `--newrelic-config-file` and `--newrelic-environment` options to `mod_wsgi-express`. This allows these to be set using command line options rather than requiring the New Relic environment variables. Importantly, when the options are used, the values will be embedded in the generated files if using `--setup-only`. Thus they will still be set when later using the `apachectl` control script to start the server.

Note that when these options are used, they will cause the equivalent New Relic environment variable for that option to be ignored, both if running the server immediately, or if using `--setup-only` and running the server later using `apachectl`.

9. Added the `--enable-debugger` option to `mod_wsgi-express`. When specified and at the same time the `--debug-mode` option is specified, then when an exception is raised from the initial execution of the WSGI application, when consuming the response iterable, or when calling any `close()` method of the response iterable, then post mortem debugging of the exception will be triggered. Post mortem debugging is performed using the Python debugger (pdb).

10. Added the `--enable-coverage` option to `mod_wsgi-express`. When specified and at the same time the `--debug-mode` option is specified, then coverage analysis is enabled. When the server is exited, then the HTML reports will be output to the `htmlcov` directory under the server working directory, or the directory specified using the `--coverage-directory` option. The `coverage` module must be installed for this feature to work.

11. Added the `--enable-profiler` option to `mod_wsgi-express`. When specified and at the same time the `--debug-mode` option is specified, then coverage analysis is enabled. When the server is exited, then the profiler data will be output to the `pstats.dat` file under the server working directory, or the file specified using the `--profiler-output-file` option.

12. Added the `--python-path` option to `mod_wsgi-express` to specify additional directories that should be added to the Python module search path.

Note that these directories will not be processed for `.pth` files. If processing of `.pth` files is required, then the `PYTHONPATH` environment variable should be set and exported in a script file referred to using the `--envvars-script` option.

Version 4.3.2

Version 4.3.2 of mod_wsgi can be obtained from:

https://codeload.github.com/GrahamDumpleton/mod_wsgi/tar.gz/4.3.2

Known Issues

1. The makefiles for building mod_wsgi on Windows are currently broken and need updating. As most new changes relate to mod_wsgi daemon mode, which is not supported under Windows, you should keep using the last available binary for version 3.X on Windows instead.

Bugs Fixed

1. Linux behaviour when using `connect()` on a non blocking UNIX socket and the listener queue is full, is apparently not POSIX compliant and it returns `EAGAIN` instead of `ECONNREFUSED`. The code handling errors from the `connect()` wasn't accomodating this non standard behaviour and so would fail immediately rather than retrying.

2. Only change working directory for mod_wsgi daemon process after having dropped privileges to target user. This is required where the specified working directory is on an NFS file system configured so as not to have root access priviliges.

3. The workaround for getting pyvenv style virtual environments to work with Python 3.3+ would break brew Python 2.7 on MacOS X as brew Python appears to not work in embedded systems which use Py_SetProgramName() instead of using Py_SetPythonHome(). Now only use Py_SetProgramName() if detect it is actually a pyvenv style virtual environment. This even appears to be okay for brew Python 3.4 at least as it does still work with the Py_SetProgramName() call even if brew Python 2.7 doesn't.

New Features

1. If the `WSGIPythonHome` directive or the `python-home` option is used with the `WSGIDaemonProcess` directive, the path provided, which is supposed to be the root directory of the Python installation or virtual environment, will be checked to see if it is actually accessible and refers to a directory. If it isn't, a warning message will be logged along with any details providing an indication of what may be wrong with the supplied path.

This is being done to warn when an invalid path has been supplied that subsequently is likely to be rejected and ignored by the Python interpreter. In such a situation where an invalid path is supplied the Python interpreter doesn't actually log anything and will instead silently fallback to using any Python installation it finds by seaching for `python` on the users `PATH`. This may not be the Python installation or virtual environment you intended be used.

2. The Apache configuration snippet generated as an example when running the `install-module` sub command of `mod_wsgi-express` to install the `mod_wsgi.so` into the Apache installation itself, will now output a `WSGIPythonHome` directive for the Python installation or virtual environment the mod_wsgi module was compiled against so that the correct Python runtime will be used.

Version 4.3.1

Version 4.3.1 of mod_wsgi can be obtained from:

https://github.com/GrahamDumpleton/mod_wsgi/archive/4.3.1.tar.gz

Known Issues

1. The makefiles for building mod_wsgi on Windows are currently broken and need updating. As most new changes relate to mod_wsgi daemon mode, which is not supported under Windows, you should keep using the last available binary for version 3.X on Windows instead.

Bugs Fixed

1. The `install-module` sub command of `mod_wsgi-express` was incorrectly trying to install the mod_wsgi `.so` file onto itself rather than into the Apache modules directory.

2. The workaround for the broken MacOS X Apache build scripts as implemented by the `configure` script used when building using the traditional make command wasn't working correctly for MacOS X 10.10 (Yosemite).

In fixing this issue, the `configure` script has been enhanced such that it is now no longer to have the whole of the Xcode package installed on MacOS X. Instead the minimum required now is the developer command line tools. If using Python and you wanted to be able to install Python packages which has a source code component you would have already likely installed the developer command line tools.

New Features

1. Added the `--add-handler` option to `mod_wsgi-express` to allow a WSGI application script file to be provided which is to handle any requests against static resources in the document root directory matching a specific extension type.

2. Added a mechanism to limit the amount of response content that can buffered in the Apache child worker processes when proxying back the response from a request which had been handled in a mod_wsgi daemon process.

This is to combat a lack of flow control within Apache 2.2 which can cause excessive amounts of memory usage as a result of such buffered content. This issue in Apache 2.2 was fixed in Apache 2.4, but the new mechanism is applied to both versions for consistency.

The default maximum on the amount of buffered content is 65536 bytes. This can be increased by using the `proxy-buffer-size` option to the `WSGIDaemonProcess` directive or the `--proxy-buffer-size` option to `mod_wsgi-express`. If using Apache 2.4, its own flow control mechanism may override the value in increasing the buffer size.

Version 4.3.0

Version 4.3.0 of mod_wsgi can be obtained from:

> https://github.com/GrahamDumpleton/mod_wsgi/archive/4.3.0.tar.gz

Known Issues

1. The makefiles for building mod_wsgi on Windows are currently broken and need updating. As most new changes relate to mod_wsgi daemon mode, which is not supported under Windows, you should keep using the last available binary for version 3.X on Windows instead.

Bugs Fixed

1. Performing authorization using the `WSGIAuthGroupScript` was not working correctly on Apache 2.4 due to changes in how auth providers and authentication/authorization works. The result could be that a user could gain access to a resource even though they were not in the required group.

2. Under Apache 2.4, when creating the `environ` dictionary for passing into access/authentication/authorisation handlers, the behvaiour of Apache 2.4 as it pertained to the WSGI application, whereby it blocked the passing of any HTTP headers with a name which did not contain just alphanumerics or '-', was not being mirrored. This created the possibility of HTTP header spoofing in certain circumstances. Such headers are now being ignored.

3. When `home` option was used with `WSGIDaemonProcess` directive an empty string was added to `sys.path`. This meant current working directory would be searched. This was fine so long as the current working directory wasn't changed, but if it was, it would no longer look in the home directory. Need to use the actual home directory instead.

4. Fixed Django management command integration so would work for versions of Django prior to 1.6 where `BASE_DIR` didn't exist in Django settings module.

Features Changed

1. In Apache 2.4, any headers with a name which does not include only alphanumerics or '-' are blocked from being passed into a WSGI application when the CGI like WSGI `environ` dictionary is created. This is a mechanism to prevent header spoofing when there are multiple headers where the only difference is the use of non alphanumerics in a specific character position.

This protection mechanism from Apache 2.4 is now being restrospectively applied even when Apache 2.2 is being used and even though Apache itself doesn't do it. This may technically result in headers that were previously being passed, no longer being passed. The change is also technically against what the HTTP RFC says is allowed for HTTP header names, but such blocking would occur in Apache 2.4 anyway due to changes in Apache. It is also understood that other web servers such as nginx also perform the same type of blocking. Reliance on HTTP headers which use characters other than alphanumerics and '-' is therefore dubious as many servers will now discard them when needing to be passed into a system which requires the headers to be passed as CGI like variables such as is the case for WSGI.

2. In Apache 2.4, only `wsgi-group` is allowed when using the `Require` directive for group authorisation. In prior Apache versions `group` would also be accepted and matched by the `wsgi` auth provider. The inability to use `group` is due to a change in Apache itself and not mod_wsgi. To avoid any issues going forward though, the mod_wsgi code will now no longer check for `group` even if for some reason Apache still decides to pass the authorisation check off to mod_wsgi even when it shouldn't.

New Features

1. The value of the `REMOTE_USER` variable for an authenticated user when user `Basic` authentication can now be overridden from an authentication handler specified using the `WSGIAuthUserScript`. To override the name used to identify the user, instead of returning `True` when indicating that the user is allowed, return the name to be used for that user as a string. That value will then be passed through in `REMOTE_USER` in place of any original value:

```
def check_password(environ, user, password):
    if user == 'spy':
        if password == 'secret':
            return 'grumpy'
        return False
    return None
```

2. Added the `--debug-mode` option to `mod_wsgi-express` which results in Apache and the WSGI application being run in a single process which is left attached to stdin/stdout of the shell where the script was run. Only a single thread will be used to handle any requests.

This feature enables the ability to interactively debug a Python WSGI application using the Python debugger (`pdb`). The simplest way to break into the Python debugger is by adding to your WSGI application code:

```
import pdb; pdb.set_trace()
```

3. Added the `--application-type` option to `mod_wsgi-express`. This defaults to `script` indicating that the target WSGI application provided to `mod_wsgi-express` is a WSGI script file defined by a relative or absolute file system path.

In addition to `script`, it is also possible to supply for the application type `module` and `paste`.

For the case of `module`, the target WSGI application will be taken to reside in a Python module with the specified name. This module will be loaded using the standard Python module import system and so must reside on the Python module search path.

For the case of `paste`, the target WSGI application will be taken to be a Paste deployment configuration file. In loading the Paste deployment configuration file, any WSGI application pipeline specified by the configuration will be constructed and the resulting top level WSGI application entry point returned used as the WSGI application.

Note that the code file for the WSGI script file, Python module, or Paste deployment configuration file, if modified, will all result in the WSGI application being automatically reloaded on the next web request.

4. Added the `--auth-user-script` and `--auth-type` options to `mod_wsgi-express` to enable the hosted site to implement user authentication using either HTTP `Basic` or `Digest` authentication mechanisms. The `check_password()` or `get_realm_hash()` functions should follow the same form as if using the `WSGIAuthUserScript` direct with mod_wsgi when using manual configuration.

5. Added the `--auth-group-script` and `--auth-group` options to `mod_wsgi-express` to enable group authorization to be performed using a group authorization script, in conjunction with a user authentication script. The `groups_for_user()` function should follow the same form as if using the `WSGIAuthGroupScript` direct with mod_wsgi when using manual configuration.

By default any users must be a member of the `wsgi` group. The name of this group though can be overridden using the `--auth-group` option. It is recommended that this be overridden rather than changing your own application to use the `wsgi` group.

6. Added the `--directory-index` option to `mod_wsgi-express` to enable a index resource to be added to the document root directory which would take precedence over the WSGI application for the root page for the site.

7. Added the `--with-php5` option to `mod_wsgi-express` to enable the concurrent hosting of a PHP web application in conjunction with the WSGI application. Due to the limitations of PHP, this is currently only supported if using prefork MPM.

8. Added the `--server-name` option to `mod_wsgi-express`. When this is used and set to the host name for the web site, a virtual host will be created to ensure that the server only accepts web requests for that host name.

If the host name starts with `www.` then web requests will also be accepted against the parent domain, that is the host name without the `www.`, but those requests will be automatically redirected to the specified host name on the same port as that used for the original request.

When the `--server-name` option is being used, the `--server-alias` option can also be specified, multiple times if need be, to setup alternate names for the web site on which web requests should also be accepted. Wildcard aliases may be used in the name if wishing to match multiple sub domains in one go.

If for some reason you do still need to be able to access the server via `localhost` when a virtual host for a set server name is being used, you can supply the `--allow-localhost` option.

9. Added the `--rotate-logs` option to `mod_wsgi-express` to enable log file rotation. By default the error log and access log, if enabled, will be rotated when they reach 5MB in size. To change the size at which the log files will be rotated, use the `--max-log-size` option. If the `rotatelogs` command is not being found properly, its location can be specified using the `--rotatelogs-executable` option.

10. Added the `--ssl-port` and `--ssl-certificate` options to `mod_wsgi-express`. When both are set, with the latter being the stub path for the SSL certificate `.crt` and `.key` file, then HTTPS requests will be handled over the designated SSL port.

When `--https-only` is supplied, any requests made over HTTP to the non SSL port will be automatically redirected so as to use a HTTPS connection over the SSL connection.

Note that if using the `--allow-localhost` option, redirection from a HTTP to HTTPS connection will not occur when access via `localhost`.

11. Added the `--setenv` option to `mod_wsgi-express` to enable request specific name/value pairs to be added to the WSGI environ dictionary. The values are restricted to string values.

Also added a companion `--passenv` option to `mod_wsgi-express` to indicate the names of normal process environment variables which should be added to the per request WSGI environ dictionary.

12. Added the `WSGIMapHEADToGET` directive for overriding the previous behaviour of automatically mapping any `HEAD` request to a `GET` request when an Apache output filter was registered that may want to see the complete response in order to generate correct response headers.

The directive can be set to be either `Auto` (the default), `On` which will always map a `HEAD` to `GET` even if no output filters detected and `Off` to always preserve the original request method type.

The original behaviour was to avoid problems with users trying to optimise for `HEAD` requests and then breaking caching mechanisms because the response headers for a `HEAD` request for a resource didn't match a `GET` request against the same resource as required by HTTP.

If using mod_wsgi-express, the `--map-head-to-get` option can be used with the same values.

12. Added the `--compress-responses` option to `mod_wsgi-express` to enable compression of common text based responses such as plain text, HTML, XML, CSS and Javascript.

Version 4.2.8

Version 4.2.8 of mod_wsgi can be obtained from:

https://github.com/GrahamDumpleton/mod_wsgi/archive/4.2.8.tar.gz

Known Issues

1. The makefiles for building mod_wsgi on Windows are currently broken and need updating. As most new changes relate to mod_wsgi daemon mode, which is not supported under Windows, you should keep using the last available binary for version 3.X on Windows instead.

Bugs Fixed

1. Disable feature for dumping stack traces on daemon process shutdown when a timeout occurs when using Python prior to 2.5. This is because the C API functions are not available in older Python versions.

2. If using Python 3.4 the minimum MacOS X version you can use is 10.8. This needs to be inforced as Apache Runtime library has a definition in header files which changes sizes from 10.7 to 10.8 and trying to compile for compatability back to 10.6 as Python 3.4 tries to enforce, will cause mod_wsgi daemon mode processes to crash at runtime.

3. Python 3.3+ pyvenv style virtual environments would not work with mod_wsgi via the `WSGIPythonHome` directive or the `home` option to the `WSGIDaemonProcess` directive. This is because the support in Python for pyvenv will not work with embedded systems which set the equivalent of `PYTHONHOME` via the Python C API.

The underlying problem in Python is described in issue:

- http://bugs.python.org/issue22213

of the Python issue tracer.

To support both normal virtualenv style virtual environments and pyvenv style virtual environments, the manner in which virtual environments are setup by mod_wsgi has been changed. This has at this point only been done on UNIX systems however, as it isn't known at this point whether the same trick will work on Windows systems.

Version 4.2.7

Version 4.2.7 of mod_wsgi can be obtained from:

https://github.com/GrahamDumpleton/mod_wsgi/archive/4.2.7.tar.gz

Known Issues

1. The makefiles for building mod_wsgi on Windows are currently broken and need updating. As most new changes relate to mod_wsgi daemon mode, which is not supported under Windows, you should keep using the last available binary for version 3.X on Windows instead.

New Features

1. Added a `--mount-point` option to `mod_wsgi-express` to allow a WSGI application to be mounted at a sub URL rather than the root of the site when using mod_wsgi express.

Version 4.2.6

Version 4.2.6 of mod_wsgi can be obtained from:

https://github.com/GrahamDumpleton/mod_wsgi/archive/4.2.6.tar.gz

Known Issues

1. The makefiles for building mod_wsgi on Windows are currently broken and need updating. As most new changes relate to mod_wsgi daemon mode, which is not supported under Windows, you should keep using the last available binary for version 3.X on Windows instead.

Bugs Fixed

1. Apache 2.2.3 and older doesn't provide the ap_get_server_description() function. Using mod_wsgi with such older versions would therefore cause processes to crash when Apache was being started up. For older versions of Apache now fallback to using ap_get_server_version() instead.

Version 4.2.5

Version 4.2.5 of mod_wsgi can be obtained from:

https://github.com/GrahamDumpleton/mod_wsgi/archive/4.2.5.tar.gz

Known Issues

1. The makefiles for building mod_wsgi on Windows are currently broken and need updating. As most new changes relate to mod_wsgi daemon mode, which is not supported under Windows, you should keep using the last available binary for version 3.X on Windows instead.

Bugs Fixed

1. When using Apache 2.4 with dynamically loaded MPM modules, mod_wsgi express was incorrectly trying to load more than one MPM module if more than one existed in the Apache modules directory.

Version 4.2.4

Version 4.2.4 of mod_wsgi can be obtained from:

https://github.com/GrahamDumpleton/mod_wsgi/archive/4.2.4.tar.gz

Known Issues

1. The makefiles for building mod_wsgi on Windows are currently broken and need updating. As most new changes relate to mod_wsgi daemon mode, which is not supported under Windows, you should keep using the last available binary for version 3.X on Windows instead.

Bugs Fixed

1. Fixed one off error in applying limit to the number of supplementary groups allowed for a daemon process group. The result could be that if more groups than the operating system allowed were specified to the option `supplementary-groups`, then memory corruption or a process crash could occur.

2. Improved error handling in setting up the current working directory and group access rights for a process when creating a daemon process group. The change means that if any error occurs that the daemon process group will be restarted rather than allow it to keep running with an incorrect working directory or group access rights.

New Features

1. Added the `--setup-only` option to mod_wsgi express so that it is possible to create the configuration when using the Django management command `runmodwsgi` without actually starting the server.

Version 4.2.3

Version 4.2.3 of mod_wsgi can be obtained from:

https://github.com/GrahamDumpleton/mod_wsgi/archive/4.2.3.tar.gz

Known Issues

1. The makefiles for building mod_wsgi on Windows are currently broken and need updating. As most new changes relate to mod_wsgi daemon mode, which is not supported under Windows, you should keep using the last available binary for version 3.X on Windows instead.

Bugs Fixed

1. The feature for starting mod_wsgi express using the Django management command `runmodwsgi` was broken by the 4.2.2 release.

Version 4.2.2

Version 4.2.2 of mod_wsgi can be obtained from:

https://github.com/GrahamDumpleton/mod_wsgi/archive/4.2.2.tar.gz

Known Issues

1. The makefiles for building mod_wsgi on Windows are currently broken and need updating. As most new changes relate to mod_wsgi daemon mode, which is not supported under Windows, you should keep using the last available binary for version 3.X on Windows instead.

Bugs Fixed

1. The `envvars` file was being overwritten even if it existed and had been modified.

New Features

1. Output the location of the `envvars` file when using the `setup-server` command for `mod_wsgi-express` or if using the `start-server` command and the `--envars-script` option was being used.

2. Output the location of the `apachectl` script when using the `setup-server` command for `mod_wsgi-express`.

Version 4.2.1

Version 4.2.1 of mod_wsgi can be obtained from:

https://github.com/GrahamDumpleton/mod_wsgi/archive/4.2.1.tar.gz

Known Issues

1. The makefiles for building mod_wsgi on Windows are currently broken and need updating. As most new changes relate to mod_wsgi daemon mode, which is not supported under Windows, you should keep using the last available binary for version 3.X on Windows instead.

Bugs Fixed

1. The auto generated configuration would not work with an Apache installation where core Apache modules were statically compiled into Apache rather than being dynamically loaded.

Version 4.2.0

Version 4.2.0 of mod_wsgi can be obtained from:

https://github.com/GrahamDumpleton/mod_wsgi/archive/4.2.0.tar.gz

Known Issues

1. The makefiles for building mod_wsgi on Windows are currently broken and need updating. As most new changes relate to mod_wsgi daemon mode, which is not supported under Windows, you should keep using the last available binary for version 3.X on Windows instead.

New Features

1. Added `mod_wsgi.server_metrics()` function which provides access to a dictionary of data derived from the Apache worker scoreboard. In effect this provides access to the same information that is used to create the Apache server status page.

Note that if `mod_status` is not loaded into Apache, or the compile time configuration of Apache prohibits the scoreboard from being available, this function will return `None`.

Also be aware that only partial information about worker status, and no information about requests, will be returned if the `ExtendedStatus` directive is not also set to `On`.

Although `mod_status` needs to be loaded, it is not necessary to enable any URL to expose the server status page.

2. Added support for a platform plugin for New Relic to `mod_wsgi-express` which will report server status information up to New Relic if the `--with-newrelic` option is supplied when running mod_wsgi express.

That same option also enables the New Relic Python agent. If you only want one or the other, you can instead use the `--with-newrelic-agent` and `--with-newrelic-platform` options.

The feature of `mod_wsgi-express` for reporting data up to the New Relic Platform is dependent upon the separate `mod_wsgi-metrics` package being installed.

Version 4.1.3

Version 4.1.3 of mod_wsgi can be obtained from:

> https://github.com/GrahamDumpleton/mod_wsgi/archive/4.1.3.tar.gz

Known Issues

1. The makefiles for building mod_wsgi on Windows are currently broken and need updating. As most new changes relate to mod_wsgi daemon mode, which is not supported under Windows, you should keep using the last available binary for version 3.X on Windows instead.

Bugs Fixed

1. The `setup.py` file wasn't always detecting the Python library version suffix properly when setting it up to be linked into the resulting `mod_wsgi.so`. This would cause an error message at link time of:

```
/usr/bin/ld: cannot find -lpython
```

Version 4.1.2

Version 4.1.2 of mod_wsgi can be obtained from:

https://github.com/GrahamDumpleton/mod_wsgi/archive/4.1.2.tar.gz

Known Issues

1. The makefiles for building mod_wsgi on Windows are currently broken and need updating. As most new changes relate to mod_wsgi daemon mode, which is not supported under Windows, you should keep using the last available binary for version 3.X on Windows instead.

Bugs Fixed

1. The integration for Django management command was looking for the wrong name for the admin script to start mod_wsgi express.

2. The code which connected to the mod_wsgi daemon process was passing an incorrect size into the connect() call for the size of the address structure. On some Linux systems this would cause an error similar to:

```
(22)Invalid argument: mod_wsgi (pid=22944): Unable to connect to \
    WSGI daemon process 'localhost:8000' on \
    '/tmp/mod_wsgi-localhost:8000:12145/wsgi.22942.0.1.sock'
```

This issue was only introduced in 4.1.0 and does not affect older versions.

3. The deadlock detection thread could try and acquire the Python GIL after the Python interpreter had been destroyed on Python shutdown resulting in the process crashing. This issue cannot be completely eliminated, but the deadlock thread will now at least check whether the flag indicating process shutdown is happening has been set before trying to acquire the Python GIL.

Version 4.1.1

Version 4.1.1 of mod_wsgi can be obtained from:

https://github.com/GrahamDumpleton/mod_wsgi/archive/4.1.1.tar.gz

Known Issues

1. The makefiles for building mod_wsgi on Windows are currently broken and need updating. As most new changes relate to mod_wsgi daemon mode, which is not supported under Windows, you should keep using the last available binary for version 3.X on Windows instead.

Bugs Fixed

1. Compilation would fail on Apache 2.4 due to a change in the Apache API to determine the name of the MPM being used.

Version 4.1.0

With version 4.1.0 of mod_wsgi, a switch to a X.Y.Z version numbering scheme from the existing X.Y scheme is being made. This is to enable a much quicker release cycle with more incremental changes.

Version 4.1.0 of mod_wsgi can be obtained from:

https://github.com/GrahamDumpleton/mod_wsgi/archive/4.1.0.tar.gz

Note that mod_wsgi 4.1.0 was originally derived from mod_wsgi 3.1. It has though all changes from later releases in the 3.X branch. Thus also see:

- *Version 3.2*
- *Version 3.3*
- *Version 3.4*
- *Version 3.5*

Known Issues

1. The makefiles for building mod_wsgi on Windows are currently broken and need updating. As most new changes relate to mod_wsgi daemon mode, which is not supported under Windows, you should keep using the last available binary for version 3.X on Windows instead.

Bugs Fixed

1. If a UNIX signal received by daemon mode process while still being initialised to signal that it should be shutdown, the process could crash rather than shutdown properly due to not registering the signal pipe prior to registering signal handler.

2. Python doesn't initialise codecs in sub interpreters automatically which in some cases could cause code running in WSGI script to fail due to lack of encoding for Unicode strings when converting them. The error message in this case was:

```
LookupError: no codec search functions registered: can't find encoding
```

The 'ascii' encoding is now forcibly loaded when initialising sub interpreters to get Python to initialise codecs.

3. Fixed reference counting bug under Python 3 in SSL `var_lookup()` function which can be used from an auth handler to look up SSL variables.

4. The `WWW-Authenticate` headers returned from a WSGI application when run under daemon mode are now always preserved as is.

Because of previously using an internal routine of Apache, way back in time the values of multiple `WWW-Authenticate` headers would be merged when there was more than one. This would cause an issue with some browsers.

A workaround was subsequently implemented above the Apache routine to break apart the merged header to create separate ones again, however, if the value of a header validly had a ',' in it, this would cause the header value to be broken apart where it wasn't meant to. This could issues with some type of `WWW-Authenticate` headers.

Features Removed

1. No longer support the use of mod_python in conjunction with mod_wsgi. When this is attempted an error is forced and Apache will not be able to start. An error message is logged in main Apache error log.

2. No longer support the use of Apache 1.3. Minimum requirement is now Apache 2.0.

Features Changed

1. Use of kernel `sendfile()` function by `wsgi.file_wrapper` is now off by default. This was originally always on for embedded mode and completely disabled for daemon mode. Use of this feature can be enabled for either mode using `WSGIEnableSendfile` directive, setting it to `On` to enable it.

The default is now off because kernel `sendfile()` is not always able to work on all file objects. Some instances where it will not work are described for the Apache `EnableSendfile` directive.

> http://httpd.apache.org/docs/2.2/mod/core.html#enablesendfile

Although Apache has use of `sendfile()` enabled by default for static files, they are moving to having it off by default in future version of Apache. This change is being made because of the problems which arise and users not knowing how to debug it and solve it.

Thus also erring on side of caution and having it off by default but allowing more knowledgeable users to enable it where they know always using file objects which will work with `sendfile()`.

2. The `HTTPS` variable is no longer set within the WSGI environment. The authoritative indicator of whether a SSL connection is used is `wsgi.url_scheme` and a WSGI compliant application should check for `wsgi.url_scheme`. The only reason that `HTTPS` was supplied at all was because early Django versions supporting WSGI interface weren't correctly using `wsgi.url_scheme`. Instead they were expecting to see `HTTPS` to exist.

This change will cause non conformant WSGI applications to finally break. This possibly includes some Django versions prior to Django version 1.0.

Note that you can still set `HTTPS` in Apache configuration using the `SetEnv` or `SetEnvIf` directive, or via a rewrite rule. In that case, that will override what `wsgi.url_scheme` is set to and once `wsgi.url_scheme` is set appropriately, the `HTTPS` variable will be removed from the set of variables passed through to the WSGI environment.

3. The `wsgi.version` variable has been reverted to 1.0 to conform to the WSGI PEP 3333 specification. It was originally set to 1.1 on expectation that revised specification would use 1.1 but that didn't come to be.

4. The `inactivity-timeout` option to `WSGIDaemonProcess` now only results in the daemon process being restarted after the idle timeout period where there are no active requests. Previously it would also interrupt a long running request. See the new `request-timeout` option for a way of interrupting long running, potentially blocked requests and restarting the process.

5. If the `home` option is used with `WSGIDaemonProcess`, in addition to that directory being made the current working directory for the process, an empty string will be added to the start of the Python module search path. This causes Python to look in the current working directory for Python modules when they are being imported.

This behaviour brings things into line with what happens when running the Python interpreter from the command line. You must though be using the `home` option for this to come into play.

Do not that if your application then changes the working directory, it will start looking in the new current working directory and not that which is specified by the `home` option. This again mirrors what the normal Python command line interpreter does.

New Features

1. Add `supplementary-groups` option to `WSGIDaemonProcess` to allow group membership to be overridden and specified comma separate list of groups used instead.

2. Add a `graceful-timeout` option to `WSGIDaemonProcess`. This option is applied in a number of circumstances.

When `maximum-requests` and this option are used together, when maximum requests is reached, rather than immediately shutdown, potentially interupting active requests if they don't finished with shutdown timeout, can specify a separate graceful shutdown period. If the all requests are completed within this time frame then will shutdown

immediately, otherwise normal forced shutdown kicks in. In some respects this is just allowing a separate shutdown timeout on cases where requests could be interrupted and could avoid it if possible.

When `cpu-time-limit` and this option are used together, when CPU time limit reached, rather than immediately shutdown, potentially interupting active requests if they don't finished with shutdown timeout, can specify a separate graceful shutdown period.

3. Add potentially graceful process restart option for daemon processes when sent a graceful restart signal. Signal is usually `SIGUSR1` but is platform dependent as using same signal as Apache would use. If the `graceful-timeout` option had been provided to `WSGIDaemonProcess`, then the process will attempt graceful shutdown first based on the that timeout, otherwise normal shutdown procedure used as if received a `SIGTERM`.

4. Add `memory-limit` option to `WSGIDaemonProcess` to allow memory usage of daemon processes to be restricted. This will have no affect on some platforms as `RLIMIT_AS`/`RLIMIT_DATA` with `setrlimit()` isn't always implemented. For example MacOS X and older Linux kernel versions do not implement this feature. You will need to test whether this feature works or not before depending on it.

5. Add `virtual-memory-limit` option to `WSGIDaemonProcess` to allow virtual memory usage of daemon processes to be restricted. This will have no affect on some platforms as `RLIMIT_VMEM` with `setrlimit()` isn't always implemented. You will need to test whether this feature works or not before depending on it.

6. Access, authentication and authorisation hooks now have additional keys in the environ dictionary for `mod_ssl.is_https` and `mod_ssl.var_lookup`. These equate to callable functions provided by `mod_ssl` for determining if the client connection to Apache used SSL and what the values of variables specified in the SSL certifcates, server or client, are. These are only available if Apache 2.0 or later is being used.

7. For Python 2.6 and above, the `WSGIDontWriteBytecode` directive can be used at global scope in Apache configuration to disable writing of all byte code files, ie., .pyc, by the Python interpreter when it imports Python code files. To disable writing of byte code files, set directive to `On`.

Note that this doesn't prevent existing byte code files on disk being used in preference to the corresponding Python code files. Thus you should first remove `.pyc` files from web application directories if relying on this option to ensure that `.py` file is always used.

8. Add `request-timeout` option to `WSGIDaemonProcess` to allow a separate timeout to be applied on how long a request is allowed to run for before the daemon process is automatically restarted to interrupt the request.

This is to counter the possibility that a request may become blocked on some backend service, thereby using up available requests threads and preventing other requests to be handled.

In the case of a single threaded process, then the timeout will happen at the specified time duration from the start of the request being handled.

Applying such a timeout in the case of a multithreaded process is more problematic as doing a restart when a single requests exceeds the timeout could unduly interfere with with requests which just commenced.

In the case of a multi threaded process, what is instead done is to take the total of the current running time of all requests and divide that by the number of threads handling requests in that process. When this average time exceeds the time specified, then the process will be restarted.

This strategy for a multithreaded process means that individual requests can actually run longer than the specified timeout and a restart will only be performed when the overall capacity of the processes appears to be getting consumed by a number of concurrent long running requests, or when a specific requests has been blocked for an excessively long time.

The intent of this is to allow the process to still keep handling requests and only perform a restart when the available capacity of the process to handle more requests looks to be potentially on the decline.

9. Add `connect-timeout` option to `WSGIDaemonProcess` to allow a timeout to be specified on how long the Apache child worker processes should wait on being able to obtain a connection to the mod_wsgi daemon process.

As UNIX domain sockets are used, connections should always succeed, however there have been some incidences seen which could only be explained by the operating system hanging on the initial connect call without being added to the daemon process socket listener queue. As such the timeout has been added. The timeout defaults to 15 seconds.

This timeout also now dictates how long the Apache child worker process will attempt to get a connection to the daemon process when the connection is refused due to the daemon socket listener queue being full. Previously how long connection attempts were tried was based on an internal retry count rather than a configurable timeout.

10. Add `socket-timeout` option to `WSGIDaemonProcess` to allow the timeout on indvidual read/writes on the socket connection between the Apache child worker and the daemon process to be specified separately to the Apache `Timeout` directive.

If this option is not specified, it will default to the value of the Apache `Timeout` directive.

11. Add `queue-timeout` option to `WSGIDaemonProcess` to allow a request to be aborted if it never got handed off to a mod_wsgi daemon process within the specified time. When this occurs a '503 Service Unavailable' response will be returned.

This is to allow one to control what to do when backlogging of requests occurs. If the daemon process is overloaded and getting behind, then it is more than likely that a user will have given up on the request anyway if they have to wait too long. This option allows you to specify that a request that was queued up waiting for too long is discarded, allowing any transient backlog to be quickly discarded and not simply cause the daemon process to become even more backlogged.

12. Add `listen-backlog` option to `WSGIDaemonProcess` to allow the daemon process socket listener backlog size to be specified. By default this limit is 100, although this is actually a hint, as different operating systems can have different limits on the maximum value or otherwise treat it in special ways.

13. Add `WSGIPythonHashSeed` directive to allow Python behaviour related to initial hash seed to be overridden when the interpreter supports it.

This is equivalent to setting the `PYTHONHASHSEED` environment variable and should be set to either `random` or a number in the range in range `[0; 4294967295]`.

14. Implemented a new streamlined way of installing mod_wsgi as a Python package using a setup.py file or from PyPi. This includes a `mod_wsgi-express` script that can then be used to start up Apache/mod_wsgi with an auto generated configuration on port 8000.

This makes it easy to run up Apache for development without interfering with the main Apache on the system and without having to worry about configuring Apache. Command line options can be used to override behaviour.

Once the `mod_wsgi` package has been installed into your Python installation, you can run:

```
mod_wsgi-express start-server
```

Then open your browser on the listed URL. This will verify that everything is working. Enter CTRL-C to exit the server and shut it down.

You can now point it at a specific WSGI application script file:

```
mod_wsgi-express start-server wsgi.py
```

For options run:

```
mod_wsgi-express start-server --help
```

If you already have another web server running on port 8000, you can override the port to be used using the `--port` option:

```
mod_wsgi-express start-server wsgi.py --port 8001
```

15. Implemented a Django application plugin to add a `runmodwsgi` command to the Django management command script. This allows the automatic run up of the new mod_wsgi express script, with it hosting the Django web site the plugin was added to.

To enable, once the `mod_wsgi` package has been installed into your Python installation, add `mod_wsgi.server` to the `INSTALLED_APPS` setting in your Django settings file.

After having run the `collectstatic` Django management command, you can then run:

```
python manage.py runmodwsgi
```

For options run:

```
python manage.py runmodwsgi --help
```

To enable automatic code reloading in a development setting, use the option:

```
python manage.py runmodwsgi --reload-on-changes
```

16. The maximum size that a response header/value can be that is returned from a WSGI application under daemon mode can now be configured. The default size has also now been increased from 8192 bytes to 32768 bytes. The name of the option to `WSGIDaemonProcess` to set the buffer size used is `header-buffer-size`.

Version 4.0

Due to version 4.0 of mod_wsgi being in development for so long, or not in development depending on how you want to look at it, and the confusion that might be caused by releasing what was sitting in the source code repository after so long, the version 4.0 moniker has been dropped. The next version after the 3.X series will therefore be version 4.1.0. With the introduction of version 4.1.0, a switch is also being made to a X.Y.Z version numbering scheme, in place of the existing X.Y version numbering scheme.

Version 3.5

Version 3.5 of mod_wsgi can be obtained from:

https://github.com/GrahamDumpleton/mod_wsgi/archive/3.5.tar.gz

Security Issues

1. Local privilege escalation when using daemon mode. (CVE-2014-0240)

The issue is believed to affect Linux systems running kernel versions >= 2.6.0 and < 3.1.0.

The issue affects all versions of mod_wsgi up to and including version 3.4.

The source of the issue derives from mod_wsgi not correctly handling Linux specific error codes from setuid(), which differ to what would be expected to be returned by UNIX systems conforming to the Open Group UNIX specification for setuid().

- http://man7.org/linux/man-pages/man2/setuid.2.html
- http://pubs.opengroup.org/onlinepubs/009695399/functions/setuid.html

This difference in behaviour between Linux and the UNIX specification was believed to have been removed in version 3.1.0 of the Linux kernel.

- https://groups.google.com/forum/?fromgroups=#!topic/linux.kernel/u6cKf4D1D-k

The issue would allow a user, where Apache is initially being started as the root user and where running code under mod_wsgi daemon mode as an unprivileged user, to manipulate the number of processes run by that user to affect the outcome of setuid() when daemon mode processes are forked and so gain escalated privileges for the users code.

Due to the nature of the issue, if you provide a service or allow untrusted users to run Python web applications you do not control the code for, and do so using daemon mode of mod_wsgi, you should update mod_wsgi as soon as possible.

Bugs Fixed

1. Python 3 installations can add a suffix to the Python library. So instead of `libpythonX.Y.so` it can be `libpythonX.Ym.so`.

2. When using daemon mode, if an uncaught exception occurred when handling a request, when response was proxied back via the Apache child process, an internal value for the HTTP status line was not cleared correctly. This was resulting in a HTTP status in response to client of '200 Error' rather than '500 Internal Server Error'.

Note that this only affected the status line and not the actual HTTP status. The status would still be 500 and the client would still interpret it as a failed request.

3. Null out Apache scoreboard handle in daemon processes for Apache 2.4 to avoid process crash when lingering close cleanup occurs.

4. Workaround broken MacOS X XCode Toolchain references in Apache apxs build configuration tool and operating system libtool script. This means it is no longer necessary to manually go into:

```
Applications/Xcode.app/Contents/Developer/Toolchains
```

and manually add symlinks to define the true location of the compiler tools.

5. Restore ability to compile mod_wsgi source code under Apache 1.3.

6. Fix checks for whether the ITK MPM is used and whether ITK MPM specific actions should be taken around the ownership of the mod_wsgi daemon process listener socket.

7. Fix issue where when using Python 3.4, mod_wsgi daemon processes would actually crash when the processes were being shutdown.

8. Made traditional library linking the default on MacOS X. If needing framework style linking for the Python framework, then use the `--enable-framework` option. The existing `--disable-framework` has now been removed given that the default action has been swapped around.

New Features

1. For Linux 2.4 and later, enable ability of daemon processes to dump core files when Apache `CoreDumpDirectory` directive used.

2. Attempt to log whether daemon process exited normally or was killed off by an unexpected signal.

Version 3.4

Version 3.4 of mod_wsgi can be obtained from:

http://modwsgi.googlecode.com/files/mod_wsgi-3.4.tar.gz

Security Issues

1. Information disclosure via Content-Type response header. (CVE-2014-0242)

The issue was identified and fixed in version 3.4 (August 2012) of mod_wsgi and is listed below at item 7 under 'Bugs Fixed'.

> Response Content-Type header could be corrupted when being sent in multithreaded configuration and embedded mode being used. Problem thus affected Windows and worker MPM on UNIX.

At the time it was believed to be relatively benign, only ever having been seen with one specific web application (Trac - http://trac.edgewall.org), with the corrupted value always appearing to be replaced with a small set of known values which themselves did not raise concerns.

A new example of this problem was identified May 2014 which opens this issue up as being able to cause arbitrary corruption of the web server HTTP response Content-Type value, resulting in possible exposure of data from the hosted web application to a HTTP client.

The new example also opens the possibility that the issue can occur with any Apache MPM and not just multithreaded MPMs as previously identified. Albeit that it still requires some form of background application threads to be in use, when a single threaded Apache MPM is being used.

In either case, it is still however restricted to the case where embedded mode of mod_wsgi is being used.

The specific scenario which can trigger the issue is where the value for the Content-Type response header is dynamically generated, and where the stack frame where the calculation was done went out of use between the time that the WSGI start_response() function was called and the first non empty byte string was yielded from the WSGI application for the response, resulting in the Python object being destroyed and memory returned to the free list.

At the same time, it would have been necessary for a parallel request thread or an application background thread to execute during that window of time and perform sufficient object allocations so as to reuse the memory previously used by the value of the Content-Type response header.

Example code which can be used to trigger the specific scenario can be found at:

https://gist.github.com/GrahamDumpleton/14b31ebe18166a89b090

That example code also provides a workaround if you find yourself affected by the issue but cannot upgrade straight away. It consists of the @intern_content_type decorator/wrapper. This can be applied to the WSGI application entry point and will use a cache to store the value of the Content-Type response header to ensure it is persistent for the life of the request.

Bugs Fixed

1. If using write() function returned by start_response() and a non string value is passed to it, then process can crash due to errors in Python object reference counting in error path of code.

2. If using write() function returned by start_response() under Python 3.X and a Unicode string is passed to it rather than a byte string, then a memory leak will occur because of errors in Python object reference counting.

3. Debug level log message about mismatch in content length generated was generated when content returned less than that specified by Content-Length response header even when exception occurring during response generation from an iterator. In the case of an exception occuring, was only meant to generate the log message if more content returned than defined by the Content-Length response header.

4. Using writelines() on wsgi.errors was failing.

5. If a UNIX signal received by daemon mode process while still being initialised to signal that it should be shutdown, the process could crash rather than shutdown properly due to not registering the signal pipe prior to registering signal handler.

6. Python doesn't initialise codecs in sub interpreters automatically which in some cases could cause code running in WSGI script to fail due to lack of encoding for Unicode strings when converting them. The error message in this case was:

```
LookupError: no codec search functions registered: can't find encoding
```

The 'ascii' encoding is now forcibly loaded when initialising sub interpreters to get Python to initialise codecs.

7. Response Content-Type header could be corrupted when being sent in multithreaded configuration and embedded mode being used. Problem thus affected Windows and worker MPM on UNIX.

Features Changed

1. The HTTPS variable is no longer set within the WSGI environment. The authoritative indicator of whether a SSL connection is used is wsgi.url_scheme and a WSGI compliant application should check for wsgi.url_scheme. The only reason that HTTPS was supplied at all was because early Django versions supporting WSGI interface weren't correctly using wsgi.url_scheme. Instead they were expecting to see HTTPS to exist.

This change will cause non conformant WSGI applications to finally break. This possibly includes some Django versions prior to Django version 1.0.

Note that you can still set HTTPS in Apache configuration using the !SetEnv or !SetEnvIf directive, or via a rewrite rule. In that case, that will override what wsgi.url_scheme is set to and once wsgi.url_scheme is set appropriately, the HTTPS variable will be removed from the set of variables passed through to the WSGI environment.

2. The wsgi.version variable has been reverted to 1.0 to conform to the WSGI PEP 3333 specification. It was originally set to 1.1 on expectation that revised specification would use 1.1 but that didn't come to be.

3. Use of kernel sendfile() function by wsgi.file_wrapper is now off by default. This was originally always on for embedded mode and completely disabled for daemon mode. Use of this feature can be enabled for either mode using WSGIEnableSendfile directive, setting it to On to enable it.

The default is now off because kernel sendfile() is not always able to work on all file objects. Some instances where it will not work are described for the Apache !EnableSendfile directive.

> http://httpd.apache.org/docs/2.2/mod/core.html#enablesendfile

Although Apache has use of sendfile() enabled by default for static files, they are moving to having it off by default in future version of Apache. This change is being made because of the problems which arise and users not knowing how to debug it and solve it.

Thus also erring on side of caution and having it off by default but allowing more knowledgeable users to enable it where they know always using file objects which will work with sendfile().

New Features

1. Support use of Python 3.2.
2. Support use of Apache 2.4.

3. Is now guaranteed that mod_ssl access handler is run before that for mod_wsgi so that any per request variables setup by mod_ssl are available in the mod_wsgi access handler as implemented by WSGIAccessScript directive.

4. Added 'python-home' option to WSGIDaemonProcess allowing a Python virtual environment to be used directly in conjunction with daemon process. Note that this option does not do anything if setting WSGILazyInitialization to 'Off'.

5. Added 'lang' and 'locale' options to WSGIDaemonProcess to perform same tasks as setting 'LANG' and 'LC_ALL environment' variables. Note that if needing to do the same for embedded mode you still need to set the environment variables in the Apache envvars file or init.d startup scripts.

6. Split combined WWW-Authenticate header returned from daemon process back into separate headers. This is work around for some browsers which require separate headers when multiple authentication providers exist.

7. For Python 2.6 and above, the WSGIDontWriteBytecode directive can be used at global scope in Apache configuration to disable writing of all byte code files, ie., .pyc, by the Python interpreter when it imports Python code files. To disable writing of byte code files, set directive to 'On'.

Note that this doesn't prevent existing byte code files on disk being used in preference to the corresponding Python code files. Thus you should first remove .pyc files from web application directories if relying on this option to ensure that .py file is always used.

8. Add supplementary-groups option to WSGIDaemonProcess to allow group membership to be overridden and specified comma separated list of groups to be used instead.

9. Add 'memory-limit' option to WSGIDaemonProcess to allow memory usage of daemon processes to be restricted. This will have no affect on some platforms as RLIMIT_AS/RLIMIT_DATA with setrlimit() isn't always implemented. For example MacOS X and older Linux kernel versions do not implement this feature. You will need to test whether this feature works or not before depending on it.

10. Add 'virtual-memory-limit' option to WSGIDaemonProcess to allow virtual memory usage of daemon processes to be restricted. This will have no affect on some platforms as RLIMIT_VMEM with setrlimit() isn't always implemented. You will need to test whether this feature works or not before depending on it.

11. Access, authentication and authorisation hooks now have additional keys in the environ dictionary for 'mod_ssl.is_https' and 'mod_ssl.var_lookup'. These equate to callable functions provided by mod_ssl for determining if the client connection to Apache used SSL and what the values of variables specified in the SSL certifcates, server or client, are. These are only available if Apache 2.0 or later is being used.

12. Add 'mod_wsgi.queue_start' attribute to WSGI environ so tools like New Relic can use it to track request queueing time. This is the time between when request accepted by Apache and when handled by WSGI application.

Version 3.3

Version 3.3 of mod_wsgi can be obtained from:

http://modwsgi.googlecode.com/files/mod_wsgi-3.3.tar.gz

Bug Fixes

1. Inactivity timeout not triggered at correct time when occurs for first request after process is started. See

http://code.google.com/p/modwsgi/issues/detail?id=182

2. Back off timer for failed connections to daemon process group wasn't working correctly and no delay on reconnect attempts was being applied. See:

http://code.google.com/p/modwsgi/issues/detail?id=195

3. Logging not appearing in Apache error log files when using daemon mode and have multiple virtual hosts against same server name. See:

http://code.google.com/p/modwsgi/issues/detail?id=204

4. Eliminate logging of !KeyError exception in threading module when processes are shutdown when using Python 2.6.5 or 3.1.2 or later. This wasn't indicating any real problem but was annoying all the same. See:

http://code.google.com/p/modwsgi/issues/detail?id=197

5. Fix potential for crash when logging error message resulting from failed group authorisation.

6. Fix compilation problems with Apache 2.3.6.

Features Changed

1. When compiled against ITK MPM for Apache, if using daemon mode, the listener socket for daemon process will be marked as being owned by the same user that daemon process runs. This will at least allow a request handled under ITK MPM to be directed to daemon process owned by same user as script. See issue:

http://code.google.com/p/modwsgi/issues/detail?id=187

2. Add isatty() to log objects used for sys.stdout/sys.stderr and wsgi.errors. The Python documentation says 'If a file-like object is not associated with a real file, this method should not be implemented'. That however is ambiguous as to whether one can omit it, or whether one should raise an NotImplementedError exception. Either way, various code doesn't cope with isatty() not existing or failing, so implement it and have it return False to be safe.

Version 3.2

Version 3.2 of mod_wsgi can be obtained from:

http://modwsgi.googlecode.com/files/mod_wsgi-3.2.tar.gz

Bug Fixes

1. The path of the handler script was reported wrongly when WSGIHandlerScript was being used and an error occurred when loading the file. Rather than the handler script file being listed, the file to which the URL mapped was reported instead.

2. Fix problem with use of condition variables/thread mutexes that was causing all requests in daemon mode on a FreeBSD system to hang immediately upon Apache being started.

http://code.google.com/p/modwsgi/issues/detail?id=176

Also use a distinct flag with condition variable in case condition variable is triggered even though condition not satisfied. This latter issue hasn't presented as a known problem, but technically a condition variable can by definition return even though not satisified. If this were to occur, undefined behaviour could result as multiple threads could listen on socket and/or accept connections on that socket at the same time.

3. Wrong check of APR_HAS_THREADS by preprocessor conditional resulting in code not compiling where APR_HAS_THREADS was defined but 0.

4. When Apache error logging redirected to syslog there is no error log associated with Apache server data structure to close. Code should always check that there is an error log to avoid crashing mod_wsgi daemon process on startup by operating on null pointer. See:

http://code.google.com/p/modwsgi/issues/detail?id=178

5. Code was not compiling with Apache 2.3. This is because ap_accept_lock_mech variable was removed. See:

http://code.google.com/p/modwsgi/issues/detail?id=186

Version 3.1

Version 3.1 of mod_wsgi can be obtained from:

http://modwsgi.googlecode.com/files/mod_wsgi-3.1.tar.gz

As this version follows on quickly from mod_wsgi 3.0, ensure you read:

- *Version 3.0*

Bug Fixes

1. Ensure that any compiler flags supplied via the CFLAGS environment variable when running 'configure' script are prefixed by '-Wc,' before being passed to 'apxs' to build module. Without this 'apxs' will incorrectly interpret the compiler options. For more details see:

http://code.google.com/p/modwsgi/issues/detail?id=166

Features Changed

1. Now give more explicit error message when compilation fails due to the Apache or Python developer header files not being installed. See:

http://code.google.com/p/modwsgi/issues/detail?id=169

Version 3.0

Version 3.0 of mod_wsgi can be obtained from:

http://modwsgi.googlecode.com/files/mod_wsgi-3.0.tar.gz

Precompiled Windows binaries for Apache 2.2 and Python 2.6 and 3.1 are also available from:

http://code.google.com/p/modwsgi/downloads/list

Note that mod_wsgi 3.0 was originally derived from mod_wsgi 2.0. It has though all changes from later releases in the 2.X branch. Thus also see:

- *Version 2.1*
- *Version 2.2*
- *Version 2.3*
- *Version 2.4*
- *Version 2.5*
- *Version 2.6*
- *Version 2.7*

Bug Fixes

1. Fix bug with quoting of options to mod_wsgi directives as described in:

http://code.google.com/p/modwsgi/issues/detail?id=55

2. For any code not run in the first Python interpreter instance, thread local data was being thrown away at the end of the request, rather than persisting through to subsequent requests handled by the same thread. This prevented caching techniques which made use of thread local storage and where data was intended to persist for the life of the process. The result was that any such data would have had to have been recreated on every request. See:

http://code.google.com/p/modwsgi/issues/detail?id=120

Features Changed

1. No longer force a zero length read before sending response headers where Apache 2.2.8 or later is used. This was originally being done as a workaround because of bug in Apache whereby it didn't generate the '100 Continue' headers properly, with possibility they would be sent as part of response content. This problem was however fixed in Apache 2.2.7 (really 2.2.8 as 2.2.7 was never publically released by ASF). Also only allow zero length read to propogate to Apache input filters when done, if the zero length read is the very first read against the input stream. For details see:

http://code.google.com/p/modwsgi/issues/detail?id=52

2. The WSGIImportScript can now appear inside of VirtualHost. However, there are now additional restrictions.

First is that the WSGIDaemonProcess directive being referred to by the WSGIImportScript directive by way of the process-group option, must appear before the WSGIImportScript directive.

Second is that the WSGIDaemonProcess directive being referred to by the WSGIImportScript directive by way of the process-group option, must appear in the same VirtualHost context, or at global server scope. It is not possible to reference a daemon process group specified in a different virtual server context.

Third is that at global server context, it is not possible to refer to a daemon process group defined in a VirtualHost context.

For additional details see:

http://code.google.com/p/modwsgi/issues/detail?id=110

3. The restriction on accessing sys.stdin and sys.stdout has been lifted. This was originally done to promote the writing of portable WSGI code. In all the campaign has failed as people can't be bothered to read the documentation to understand why it was done and instead use the workaround and don't actually fix the code that isn't portable. More details at:

http://blog.dscpl.com.au/2009/04/wsgi-and-printing-to-standard-output.html

4. Reenabled WSGIPythonHome directive in Windows as does apparently work so long as virtual environment setup correctly for it to refer to.

5. WSGI version now marked as WSGI 1.1 instead of 1.0. This is on basis that proposed ammendments to WSGI which mod_wsgi already implements will at least be accepted as WSGI 1.1 independent of any discussions of changing WSGI interface to use unicode with encoding other than Latin-1.

6. Set timeout on socket connection between Apache server child process and daemon process earlier to catch any blocking problems in initial handshake between the processes. This will make code more tolerant of any unexpected issues with socket communications.

Features Removed

1. The WSGIReloadMechanism directive has been removed. This means that script reloading is not available as an option in daemon mode and the prior default of process reloading always used, unless of course WSGIScriptReloadig is Off and all reloading is disabled. Doesn't affect embedded mode where script reloading was always the only option. For details see:

http://code.google.com/p/modwsgi/issues/detail?id=72

2. There is no longer an attempt to set Content-Length header for a response if not supplied and iterable was a sequence of length 1. This was suggested by WSGI specification but turns out this causes problems with HEAD requests. For details see:

http://blog.dscpl.com.au/2009/10/wsgi-issues-with-http-head-requests.html

Note that Apache may still do the same thing in certain circumstances. Whether Apache always does the correct thing is not known.

In general, a WSGI application should always return full response content for a HEAD request and should NOT truncate the response.

Features Added

1. Support added for using Python 3.X.

What constitutes support for Python 3.X is described in:

http://code.google.com/p/modwsgi/wiki/SupportForPython3X

Note that Python 3.0 is not supported and cannot be used. You must use Python 3.1 or later as mod_wsgi relies on features only added in Python 3.1. The PSF has also affectively abandoned Python 3.0 now anyway.

Also note that there is no official WSGI specification for Python 3.X and objections could be raised about what mod_wsgi has implemented. If that occurs then mod_wsgi may need to stop claiming to be WSGI compliant.

2. It is now possible to supply 'process-group', 'application-group', 'callable-object' and 'pass-authorization' configuration options to the WSGIScriptAlias and WSGIScriptAliasMatch directives after the location of the WSGI script file parameter. For example:

```
WSGIScriptAlias /trac /var/trac/apache/trac.wsgi \
 process-group=trac-projects application-group=%{GLOBAL}
```

Where the options are provided, these will take precedence over any which apply to the application as defined in Location or Directory configuration containers.

For WSGIScriptAlias (but not WSGIScriptAliasMatch) where both 'process-group' and 'application-group' parameters are provided, and neither use expansion variables that can only be evaluated at the time of request handling, this will also cause the WSGI script file to be preloaded when the process starts, rather than being lazily loaded only when first request for application arrives.

Preloading of the WSGI script is performed in the same way as when using the WSGIImportScript directive. The above configuration is therefore equivalent to existing, but longer way of doing it, as shown below:

```
WSGIScriptAlias /trac /var/trac/apache/trac.wsgi

WSGIImportScript /var/trac/apache/trac.wsgi \
 process-group=trac-projects application-group=%{GLOBAL}

<Directory /var/trac/apache>
WSGIProcessGroup trac-projects
WSGIApplicationGroup %{GLOBAL}
</Directory>
```

Note that the WSGIDaemonProcess directive defining the daemon process group being referred to by the process-group option must preceed the WSGIScriptAlias directive in the configuration file. Further, you can only refer to a daemon process group referred to in the same VirtualHost context, or at global server scope.

3. When client closes connection and iterable returned from WSGI application being processed, now directly log message at debug level in log files, rather than raising a Python exception and with that being logged at error level as was previously the case.

For where write() being called a Python exception still has to be raised and whether that results in any message being logged depends on what the WSGI application does.

End result is that for normal case where LogLevel wouldn't be set to debug, the log file will not fill up with messages where client prematurely closes connection.

For details see:

http://code.google.com/p/modwsgi/issues/detail?id=29

4. Added new 'chroot' option to WSGIDaemonProcess directive to force daemon process to run inside of a chroot environment.

For this to work you need to have a working Python installation installed into the chroot environment such that inside of that context it appears at same location as that which Apache/mod_wsgi is running.

Note that the WSGI application code and any files it require have to be located within the chroot directory structure. In configuring mod_wsgi reference is then made to the WSGI application at that location. Thus:

```
WSGIDaemonProcess choot-1 user=grahamd group=staff display-name=%{GROUP} \
    root=/some/path/chroot-1

WSGIScriptAlias /app /some/path/chroot-1/var/www/app/scripts/app.wsgi \
    process-group=chroot-1
```

Normally this would result in Apache generating SCRIPT_FILENAME as the path as second argument to WSGIScriptAlias, but mod_wsgi, knowing it is a chroot environment will adjust that path and drop the chroot directory root from front of path so that it resolves correctly when used in context of chroot environmet.

In other words, there is no need to create a parallel directory structure outside of chroot environment just to satisfy Apache URL mapper.

Any static files can be in or outside of the chroot directory and will still be served by Apache child worker processes, which don't run in chroot environment. If user only has access to chroot environment through login shell that goes directly to it, then static files will obviously be inside.

How to create a chroot environment will not be described here and you will want to know what you are doing if you want to use this feature. For some pointers to what may need to be done for Debian/Ubuntu see article at:

http://transcyberia.info/archives/12-chroot-plone-buildouts.html

For details on this change also see:

http://code.google.com/p/modwsgi/issues/detail?id=106

5. Added WSGIPy3kWarningFlag directive when Python 2.6 being used. This should be at server scope outside of any VirtualHost and will apply to whole server:

```
WSGIPy3kWarningFlag On
```

This should have same affect as -3 option to 'python' executable. For more details see:

http://code.google.com/p/modwsgi/issues/detail?id=109

6: Fix up how Python thread state API is used to avoid internal Python assertion error when Python compiled with Py_DEBUG preprocessor symbol. For details see:

http://code.google.com/p/modwsgi/issues/detail?id=113

7. Now allow chunked request content. Such content will be dechunked and available for reading by WSGI application. See:

> http://code.google.com/p/modwsgi/issues/detail?id=1

To enable this feature, you must use:

```
WSGIChunkedRequest On
```

for appropriate context in Apache configuration.

Do note however that WSGI is technically incapable of supporting chunked request content without all chunked request content having to be first read in and buffered. This is because WSGI requires CONTENT_LENGTH be set when there is any request content.

In mod_wsgi no buffering is done. Thus, to be able to read the request content in the case of a chunked transfer encoding, you need to step outside of the WSGI specification and do things it says you aren't meant to.

You have two choices for how you can do this. The first choice you have is to call read() on wsgi.input but not supply any argument at all. This will cause all request content to be read in and returned.

The second is to loop on calling read() on wsgi.input with a set block size passed as argument and do this until read() returns an empty string.

Because both calling methods are not allowed under WSGI specification, in using these your code will not be portable to other WSGI hosting mechanisms.

8. Values for HTTP headers now passed in environment dictionary to access, authentication and authorisation hooks. See:

> http://code.google.com/p/modwsgi/issues/detail?id=69

9. The flag wsgi.run_once is not set to True when running in daemon mode and both threads and maximum-requests is set to 1. With this configuration, are gauranteed that process will only be used once before being restarted. Note that don't get this gaurantee when multiple threads used as the maximum requests is only checked at end of successful request and so could feasibly still have multiple concurrent requests in progress at that point and so process wasn't used only once.

10. Added lazy initialisation of Python interpreter. That is, Python interpreter will not be initialised in Apache parent process and inherited across fork when creating child processes. Instead, the Python interpreter will only first be initialised in child process after the fork.

This behaviour is now the default as Python 3.X by design doesn't cleanup memory when interpreter destroyed. This causes significant memory leaks into Apache parent process as not reclaiming the memory doesn't work well with fact that Apache will unload Python library on an Apache restart and loose references to that unclaimed memory, such that when Python is reinitialised, it can't reuse it.

In Python 2.X it does attempt to reclaim all memory when Python interpreter is destroyed, but some Python versions still leak some memory due to real leaks or also perhaps by design as per Python 3.X. In Python 2.X the leaks are far less significant and have been tolerated in the past. The leaks in Python 2.X only cause problems if you do lots of Apache restarts, rather than stop/start. All the same, default for Python 2.X has also now been made to perform lazy initialisation.

To control the behaviour have added the directive WSGILazyInitialization. This defaults to On for both Python 2.X and Python 3.X. If you wish to experiment with whether early initialisation gives better results for Python 2.X, you can set this directive to Off.

The downside of performing lazy initialisation is that you may loose some benefit of being able to share memory between child process. Thus, child processes will potentially consume more resident memory than before due to data being local to process rather than potentially being shared.

If you are exclusively using mod_wsgi daemon mode and not using embedded mode, if lazy initialisation is used in conjunction with WSGIRestrictEmbedded being set to On, then the Python interpreter will not be initialised at all in the Apache server child processes, unless authentication providers or other non content generation code is being provided to be executed in Apache server child processes. This means that Apache worker processes will be much smaller.

Even when initialisation of Python in Apache worker processes is disabled, as before, the mod_wsgi daemon processes will still use more resident memory over shared memory. If however you are only running a small number of mod_wsgi daemon processes, then this may overall balance out as using less memory in total.

For more details see:

http://code.google.com/p/modwsgi/issues/detail?id=99

11. If daemon process defined in virtual host which has its own error log, then associated stderr with that virtual hosts error log instead. This way any messages sent direct to stderr from C extension modules will end up in the virtual host error log that the daemon process is associated with, rather than the main error log.

12. If daemon process defined in a virtual host, close all error logs for other virtual hosts which don't reference the same error log. This ensures that code can't write messages to error logs for another host, or reopen the log and read data from the logs.

13. Implement internal server redirection using Location response header as allowed for in CGI specification. Note though that this feature has only been implemented for mod_wsgi daemon mode. See:

http://code.google.com/p/modwsgi/issues/detail?id=14

14. Implement WSGIErrorOverride directive which when set to On will result in Apache error documents being used rather than those passed back by the WSGI application. This allows error documents to match any web site that the WSGI application may be integrated as a part of. This feature is akin to the ProxyErrorOverride directive of Apache but for mod_wsgi only. Do note though that this feature has only been implemented for mod_wsgi daemon mode. See:

http://code.google.com/p/modwsgi/issues/detail?id=57

15. Implement WSGIPythonWarnings directive as equivalent to the 'python' executable '-W' option. The directive can be used at global scope in Apache configuration to provide warning control strings to disable messages produced by the warnings module. For example:

```
# Ignore everything.
WSGIPythonWarnings ignore
```

or:

```
# Ignore only DeprecationWarning.
WSGIPythonWarnings ignore::DeprecationWarning::
```

For more details see:

http://code.google.com/p/modwsgi/issues/detail?id=137

16. Added cpu-time-limit option to WSGIDaemonProcess directive. This allows one to define a time in seconds which will be the maximum amount of cpu time the process is allowed to use before a shutdown is triggered and the daemon process restarted. The point of this is to provide some means of controlling potentially run away processes due to bad code that gets stuck in heavy processing loops. For more details see:

http://code.google.com/p/modwsgi/issues/detail?id=21

17. Added cpu-priority option to WSGIDaemonProcess directive. This allows one to adjust the CPU priority associated with processes in a daemon process groups. The range of values that can be supplied is dictated by what the setpriority() function on your particular operating system accepts. Normally this is in the range of about -20 to 20, with 0 being normal. For more details see:

http://code.google.com/p/modwsgi/issues/detail?id=142

18. Added WSGIHandlerScript directive. This allows one to nominate a WSGI script file that should be executed as a handler for a specific file type as configured within Apache. For example:

```
<Files *.bobo>
WSGIProcessGroup bobo
WSGIApplicationGroup %{GLOBAL}
MultiViewsMatch Handlers
Options +ExecCGI
</Files>
AddHandler bobo-script .bobo
WSGIHandlerScript bobo-script /some/path/bobo-handler/handler.wsgi
```

For this example, the application within the WSGI script file will be invoked whenever a URL maps to a file with '.bobo' extension. The name of the file mapped to by the URL will be available in the 'SCRIPT_FILENAME' WSGI environment variable.

Although same calling interface is used as a WSGI application, to distinguish that this is acted as a handler, the application entry point must be called 'handle_request' and not 'application'.

When providing such a handler script, it is also possible to provide in the script file a 'reload_required' callable object. This will be called prior to handling a request and allows the script to determine if a reload should be performed first. In the case of daemon mode, this allows script to programmatically determine if the whole process should be reloaded first. The argument to the 'reload_required' function is the original resource file that was the target of the request and which would have been available to the handler as SCRIPT_FILENAME.

Version 2.8

Version 2.8 of mod_wsgi can be obtained from:

http://modwsgi.googlecode.com/files/mod_wsgi-2.8.tar.gz

Bug Fixes

1. Ensure that any compiler flags supplied via the CFLAGS environment variable when running 'configure' script are prefixed by '-Wc,' before being passed to 'apxs' to build module. Without this 'apxs' will incorrectly interpret the compiler options. For more details see:

http://code.google.com/p/modwsgi/issues/detail?id=166

Version 2.7

Version 2.7 of mod_wsgi can be obtained from:

http://modwsgi.googlecode.com/files/mod_wsgi-2.7.tar.gz

Note that this release does not support Python 3.0. Python 3.0 will only be supported in mod_wsgi 3.0.

Features Changed

1. Set timeout on socket connection between Apache server child process and daemon process earlier to catch any blocking problems in initial handshake between the processes. This will make code more tolerant of any unexpected

issues with socket communications.

Bug Fixes

1. Wasn't possible to set CFLAGS from environment variable when running the 'configure' script.

Version 2.6

Version 2.6 of mod_wsgi can be obtained from:

http://modwsgi.googlecode.com/files/mod_wsgi-2.6.tar.gz

For Windows binaries see:

http://code.google.com/p/modwsgi/wiki/InstallationOnWindows

Note that this release does not support Python 3.0. Python 3.0 will only be supported in mod_wsgi 3.0.

Note that the fix for (3) below is believed to have already been backported to mod_wsgi 2.5 in Debian Stable tree. Thus, if using mod_wsgi 2.5 from Debian you do not need to be concerned about upgrading to this version.

Bug Fixes

1. Fixed build issue on MacOS X where incorrect Python framework found at run time. This was caused by '-W,-l' option prefix being dropped from '-F' option in LDFLAGS of Makefile and not reverted back when related changes undone. This would affect Python 2.3 through 2.5. For more details see:

http://code.google.com/p/modwsgi/issues/detail?id=28

2. Fixed build issue on MacOS X where incorrect Python framework found at run time. This was caused by '-L/-l' flags being used for versions of Python prior to 2.6. That approach, even where '.a' library link to framework exists, doesn't seem to work for the older Python versions.

Because of the unpredictability as to when '-F/-framework' or '-L/-l' should be used for specific Python versions or distributions. Now always link against Python framework via '-F/-framework' if available. If for some particular setup this isn't working, then the '–disable-framework' option can be supplied to 'configure' script to force use of '-L/-l'. For more details see:

http://code.google.com/p/modwsgi/issues/detail?id=28

3. Fixed bug where was decrementing Python object reference count on NULL pointer, causing a crash. This was possibly only occuring in embedded mode and only where closure of remote client connection was detected before any request content was read. The issue may have been more prevalent for a HTTPS connection from client.

4. Fixed bug for Python 2.X where when using 'print' to output multple objects to log object via, wsgi.errors, stderr or stdout, a space wasn't added to output between objects. This was occuring because log object lacked a softspace attribute.

Features Changed

1. When trying to determining version of Apache being used at build time, if Apache executable not available, fallback to getting version from the installed Apache header files. Do this as some Linux distributions build boxes do not actually have Apache executable itself installed, only the header files and apxs tool needed to build modules. For more details see:

http://code.google.com/p/modwsgi/issues/detail?id=147

Version 2.5

Version 2.5 of mod_wsgi can be obtained from:

http://modwsgi.googlecode.com/files/mod_wsgi-2.5.tar.gz

For Windows binaries see:

http://code.google.com/p/modwsgi/wiki/InstallationOnWindows

Note that this release does not support Python 3.0. Python 3.0 will only be supported in mod_wsgi 3.0.

Bug Fixes

1. Change to workaround problem where correct version of Python framework isn't being found at run time and instead uses the standard system one, which may be the wrong version. Change is for those Python versions on MacOS X which include a .a in Python config directory, which should be symlinked to framework, link against the .a instead. For some reason, doing this results in framework then being picked up from the correct location.

This problem may well have only started cropping up at some point due to a MacOS X Leopard patch update as has been noticed that Python frameworks installed previously stopped being found properly when mod_wsgi was subsequently recompiled against them. Something may therefore have changed in compiler tools suite.

For more details see:

http://code.google.com/p/modwsgi/issues/detail?id=28

2. Remove isatty from Log object used for stdout/stderr. It should have been a function and not an attribute. Even so, isatty() is not meant to be supplied by a file like object if it is associated with a file descriptor. Thus, packages which want to use isatty() are supposed to check for its existance before calling it. Thus wasn't ever mod_wsgi that was wrong in not supply this, but the packages which were trying to use it.

For more details see:

http://code.google.com/p/modwsgi/issues/detail?id=146

Version 2.4

Version 2.4 of mod_wsgi can be obtained from:

http://modwsgi.googlecode.com/files/mod_wsgi-2.4.tar.gz

Bug Fixes

1. Compilation would fail on Windows due to daemon mode specific code not being conditionally compiled out on that platform. This was a problem introduced by changes in mod_wsgi 2.3.

2. Fix bug where wrong Apache memory pool used when processing configuration directives at startup. This could later result in memory corruption and may account for problems seen with 'fopen()' errors. See:

http://code.google.com/p/modwsgi/issues/detail?id=78

http://code.google.com/p/modwsgi/issues/detail?id=108

3. Fix bug where Python interpreter not being destroyed correctly in Apache parent process on an Apache restart. This was resulting in slow memory leak into Apache parent process on each restart. This additional memory usage would then be inherited by all child processes forked from Apache parent process.

Note that this change does not help for case where mod_python is also being loaded into Apache as in that case mod_python is responsible for intialising Python and in all available versions of mod_python it still doesn't properly destroy the Python interpreter either and so causes memory leaks which mod_wsgi cannot work around.

Also, this doesn't solve problems with the Python interpreter itself leaking memory when destroyed and reinitialised. Such memory leaks in Python seem to occur for some versions of Python on particular platforms.

For further details see:

http://code.google.com/p/modwsgi/issues/detail?id=99

4. Fix bug whereby POST requests where 100-continue was expected by client would see request content actually truncated and not be available to WSGI application if application running in daemon mode. See:

http://code.google.com/p/modwsgi/issues/detail?id=121

5. Fix bug where Apache optimisation related to keep alive connections can kick in when using wsgi.file_wrapper with result that if amount of data is between 255 and aproximately 8000 bytes, that a completely empty response will result. This occurs because Apache isn't flushing out the file data straight away but holding it over in case subsequent request on connection arrives. By then the file object used with wsgi.file_wrapper can have been closed and underlying file descriptor will not longer be valid. See:

http://code.google.com/p/modwsgi/issues/detail?id=132

6. Modify how daemon process shutdown request is detected such that no need to block signals in request threads. Doing this caused problems in processes which were run from daemon mode process and which needed to be able to receive signals. New mechanism uses a internal pipe to which signal handler writes a character, with main thread performing a poll on pipe waiting for that character to know when to shutdown. For additional details see:

http://code.google.com/p/modwsgi/issues/detail?id=87

7. Fix bug where excessive transient memory usage could occur when calling read() or readline() on wsgi.input with no argument. See:

http://code.google.com/p/modwsgi/issues/detail?id=126

Note that calling read() with no argument is actually a violation of WSGI specification and any application doing that is not a WSGI compliant application.

8. Fix bug where daemon process would crash if User/Group directives were not specified prior to WSGIDaemonProcess in Apache configuration file. See:

http://code.google.com/p/modwsgi/issues/detail?id=40

9. Fix bug whereby Python exception state wasn't being cleared correctly when error occurred in loading target of WSGIImportScript. See:

http://code.google.com/p/modwsgi/issues/detail?id=117

Features Changed

1. No longer populate 'error-notes' field in Apache request object notes table, with details of why WSGI script failed. This has been removed as information can be seen in default Apache multilanguage error documents. Because errors may list paths or user/group information, could be seen as a security risk.

Features Added

1. Added 'mod_wsgi.version' to WSGI environment passed to WSGI application. For details see:

http://code.google.com/p/modwsgi/issues/detail?id=93

2. Added 'process_group' and 'application_group' attributes to mod_wsgi module that is created within each Python interpreter instance. This allows code executed outside of the context of a request handler to know whether it is running in a daemon process group and what it may be called. Similarly, can determine if running in first interpreter or some other sub interpreter. For details see:

http://code.google.com/p/modwsgi/issues/detail?id=27

3. Added closed and isatty attributes to Log object as well as close() method. For wsgi.errors these aren't required, but log object also used for stderr and stdout (when enabled) and code may assume these methods may exist for stderr and stdout. The closed and isatty attributes always yield false and close() will raise a run time error indicating that log cannot be closed. For details see:

http://code.google.com/p/modwsgi/issues/detail?id=82

4. Apache scoreboard cleaned up when daemon processes first initialised to prevent any user code interfering with operation of Apache. For details see:

http://code.google.com/p/modwsgi/issues/detail?id=104

5. When running configure script, can now supply additional options for CPPFLAGS, LDFLAGS and LDLIBS through environment variables. For details see:

http://code.google.com/p/modwsgi/issues/detail?id=107

6. Better checking done on response headers and an explicit error will now be produce if name or value of response header contains an embedded newline. This is done as by allowing embedded newline would cause daemon mode to fail when handing response in Apache child process. In embedded mode, could allow application to pass back malformed response headers to client. For details see:

http://code.google.com/p/modwsgi/issues/detail?id=81

7: Ensure that SYSLIBS linker options from Python configuration used when linking mod_wsgi Apache module. This is now prooving necessary as some Apache distributions are no longer linking system maths library and Python requires it. To avoid problem simply link against mod_wsgi Apache module and system libraries that Python needs. For details see:

http://code.google.com/p/modwsgi/issues/detail?id=115

8: Reorder sys.path after having called site.addsitedir() in WSGIPythonPath and python-path option for WSGIDaemonProcess. This ensures that newly added directories get moved to front of sys.path and that they take precedence over standard directories. This in part avoids need to ensure –no-site-packages option used when creating virtual environments, as shouldn't have an issue with standard directories still overriding additions. For details see:

http://code.google.com/p/modwsgi/issues/detail?id=112

9. Update USER, USERNAME and LOGNAME environment variables if set in daemon process to be the actual user that the process runs as rather than what may be inherited from Apache root process, which would typically be 'root' or the user that executed 'sudo' to start Apache, if they hadn't used '-H' option to 'sudo'. See:

http://code.google.com/p/modwsgi/issues/detail?id=129

10. Build process now inserts what is believed to be the directory where Python shared library is installed, into the library search path before the Python config directory. This should negate the need to ensure that Python shared library is also symlink into the config directory next to the static library as linkers would normally expect it. See:

http://code.google.com/p/modwsgi/issues/detail?id=136

Version 2.3

Version 2.3 of mod_wsgi can be obtained from:

http://modwsgi.googlecode.com/files/mod_wsgi-2.3.tar.gz

Note that this is a quick followup to version 2.2 of mod_wsgi to rectify significant problem introduced by that release. You should therefore also refer to:

- *Version 2.2*

Bug Fixes

1. Fixed problem introduced in version 2.2 of mod_wsgi whereby use of daemon mode would cause CGI scripts to fail.

It is quite possible that the bug could also have caused failures with other Apache modules that relied on registering of cleanup functions against Apache configuration memory pool.

For details see:

http://groups.google.com/group/modwsgi/browse_frm/thread/79a86f8faffe7dcf

2. When using setproctitle() on BSD systems, first argument should be a printf style format string with values to fill out per format as additional arguments. Code was supplying value to be displayed as format string which meant that if it contained any printf type format sequences, could cause process to crash as corresponding arguments wouldn't have ben provided.

For details see:

http://code.google.com/p/modwsgi/issues/detail?id=90

Version 2.2

Version 2.2 of mod_wsgi can be obtained from:

http://modwsgi.googlecode.com/files/mod_wsgi-2.2.tar.gz

Note: This version was quickly superseded by version 2.3 of mod_wsgi. Version 2.2 should not be used.

Features Changed

1. Use official way of setting process names on FreeBSD, NetBSD and OpenBSD. For details see:

http://code.google.com/p/modwsgi/issues/detail?id=90

This is a backport of change from version 3.0 of mod_wsgi.

Bug Fixes

1. Fix bug whereby if mod_python is loaded at same time as mod_wsgi the WSGIImportScript directive can cause Apache child processes to crash. For details see:

http://code.google.com/p/modwsgi/issues/detail?id=91

2. Fix bug where mod_wsgi daemon process startup could fail due to old stale UNIX listener socket file as described in:

http://code.google.com/p/modwsgi/issues/detail?id=77

3. Fix bug where listener socket file descriptors for daemon processes were being leaked in Apache parent process on a graceful restart. Also fixes problem where UNIX listener socket was left in filesystem on both graceful restart and graceful shutdown. For details see:

http://code.google.com/p/modwsgi/issues/detail?id=95

4. Fix bug where response was truncated when a null character appeared as first character in block of data being returned from wsgi.file_wrapper. Only occurred when code fell back to using iteration over supplied file like object, rather than optimised method such as sendfile().

http://code.google.com/p/modwsgi/issues/detail?id=100

Version 2.1

Version 2.1 of mod_wsgi can be obtained from:

http://modwsgi.googlecode.com/files/mod_wsgi-2.1.tar.gz

Bug Fixes

1. Fix bug which was resulting in logging destined for !VirtualHost !ErrorLog going missing or ending up in main Apache error log.

http://code.google.com/p/modwsgi/issues/detail?id=79

2. Fix bug where WSGI application returning None rather than valid iterable causes process to crash.

http://code.google.com/p/modwsgi/issues/detail?id=88

Version 2.0

Version 2.0 of mod_wsgi can be obtained from:

http://modwsgi.googlecode.com/files/mod_wsgi-2.0.tar.gz

Note that mod_wsgi 2.0 was originally derived from mod_wsgi 1.0. It has though all changes from later releases in the 1.X branch. Thus also see:

- *Version 1.1*
- *Version 1.2*
- *Version 1.3*

Bug Fixes

1. Work around bug in Apache where '100 Continue' response was sent as part of response content if no attempt to read request input before headers and response were generated.

Features Changed

1. The WSGICaseSensitivity directive can now only be used at global scope within the Apache configuration. This means that individual directories can not be designated as being case sensitive or not. For correct operation therefore, the path names of all script files should treat case the same, one cannot have a mixture.

2. How the WSGIPythonPath directive is interpreted has changed in that '.pth' files in the desiginated directories are honoured. See item 10 in new features section for more information.

3. Removed support for output buffering outside of WSGI specification. In other words, removed the WSGIOutputBuffering directive and associated code. If using a WSGI application which does poor buffering itself, to the extent that performance is affected, you will need to wrap it in a WSGI middleware component that does buffering on its behalf.

Features Removed

1. The 'Interpreter' option to WSGIReloadMechanism has been removed. This option for interpreter reloading was of limited practical value as many third party modules for Python aren't written in a way to cope with destruction of Python interpreters in a running process. The presence of the feature was just making it harder to implement various new features.

2. The WSGIPythonHome directive is no longer available on Windows systems as Python would ignore it anyway.

3. The WSGIPythonExecutable directive has been removed. This didn't work on Windows or MacOS X systems. On UNIX systems, the WSGIPythonHome directive should be used instead. Not known how one can achieve same on Windows systems.

Features Added

1. The WSGIReloadMechanism now provides the 'Process' option for enabling process reloading when the WSGI script file is changed. Note that this only applies to WSGI script files used for WSGI applications which have been delegated to a mod_wsgi daemon process. Additionally, as of 2.0c5 the use of 'Process' option has been made the default for daemon mode processes. If specifically requiring existing default behaviour, the 'Module' option will need to be specified to indicate script file reloading.

If this option is specified for WSGI application run in embedded mode within Apache child processes, the existing default behaviour of reloading just the script file will apply.

For more details see:

> http://code.google.com/p/modwsgi/wiki/ReloadingSourceCode

2. When application is running in embedded mode, and WSGIApacheExtensions directive is set to On, then a Python CObject reference is added to the WSGI application environment as 'apache.request_rec'. This can be passed to C extension modules and can be converted back to a reference to internal Apache request_rec structure thereby allow C extension modules to work against the internal Apache C APIs to implement special features.

One example of such special extensions are the Python SWIG bindings for the Apache C API implemented in the separate 'ap_swig_py' package. Because SWIG is being used, and due to thread support within SWIG generated bindings possibly only being usable within the first Python interpreter instance created, it may be the case that the 'ap_swig_py' package an only be used when WSGIApplicationGroup has been set to '%{GLOBAL}'.

The 'ap_swig_py' package has not yet been released and is still in development. The package can be obtained from the Subversion repository at:

> https://bitbucket.org/grahamdumpleton/apswigpy/wiki/Home

With the SWIG binding for the Apache API, the intention is that many of the internal features of Apache would then be available. For example:

```
import apache.httpd, apache.http_core

req = apache.httpd.request_rec(environ["apache.request_rec"])
root = apache.http_core.ap_document_root(req)
```

Note that this feature is experimental and may be removed from a future version if insufficient interest in it or in developing SWIG bindings.

3. When Apache 2.0/2.2 is being used, Python script can now be provided to perform the role of an Apache auth provider. This would allow user authentication underlying HTTP Basic (2.0 and 2.2) or Digest (2.2 only) authentication schemes to be done by a Python web application. Do note though that at present the provided authentication script will always run in the context of the Apache child processes and can not be delegated to a distinct daemon process.

Apache configuration for defining an auth provider for Basic authentication when using Apache 2.2 would be:

```
AuthType Basic
AuthName "Top Secret"
AuthBasicProvider wsgi
WSGIAuthUserScript /usr/local/wsgi/scripts/auth.wsgi
Require valid-user
```

For Apache 2.0 it would be:

```
AuthType Basic
AuthName "Top Secret"
WSGIAuthUserScript /usr/local/wsgi/scripts/auth.wsgi
AuthAuthoritative Off
Require valid-user
```

The 'auth.wsgi' script would then need to contain a 'check_password()' function with a sample as shown below:

```
def check_password(environ, user, password):
    if user == 'spy':
        if password == 'secret':
            return True
        return False
    return None
```

If using Apache 2.2 and Digest authentication support is built into Apache, then that also may be used:

```
AuthType Digest
AuthName "Top Secret"
AuthDigestProvider wsgi
WSGIAuthUserScript /usr/local/wsgi/scripts/auth.wsgi
Require valid-user
```

The name of the required authentication function for Digest authentication is 'get_realm_hash()'. The result of the function must be 'None' if the user doesn't exist, or a hash string encoding the user name, authentication realm and password:

```
import md5

def get_realm_hash(environ, user, realm):
    if user == 'spy':
        value = md5.new()
        # user:realm:password
```

```
        value.update('%s:%s:%s' % (user, realm, 'secret'))
        hash = value.hexdigest()
        return hash
    return None
```

By default the auth providers are executed in context of first interpreter created by Python. This can be overridden using the 'application-group' option to the script directive. The namespace for authentication groups is shared with that for application groups defined by WSGIApplicationGroup.

If mod_authn_alias is being loaded into Apache, then an aliased auth provider can also be defined:

```
<AuthnProviderAlias wsgi django>
WSGIAuthUserScript /usr/local/django/mysite/apache/auth.wsgi \
 application-group=django
</AuthnProviderAlias>

WSGIScriptAlias / /usr/local/django/mysite/apache/django.wsgi

<Directory /usr/local/django/mysite/apache>
Order deny,allow
Allow from all

WSGIApplicationGroup django

AuthType Basic
AuthName "Django Site"
AuthBasicProvider django
Require valid-user
</Directory>
```

An authentication script for Django might then be something like:

```
import os, sys
sys.path.append('/usr/local/django')
os.environ['DJANGO_SETTINGS_MODULE'] = 'mysite.settings'

from django.contrib.auth.models import User
from django import db

def check_password(environ, user, password):
    db.reset_queries()

    kwargs = {'username': user, 'is_active': True}

    try:
        try:
            user = User.objects.get(**kwargs)
        except User.DoesNotExist:
            return None

        if user.check_password(password):
            return True
        else:
            return False
    finally:
        db.connection.close()
```

If the WSGIApacheExtensions directive is set to On then 'apache.request_rec' will be passed in 'environ' to the auth

provider functions. This may be used in conjunction with C extension modules such as 'ap_swig_py'. For example, it may be used to set attributes in 'req.subprocess_env' which are then in turn passed to the WSGI application through the WSGI environment. Passing of these settings will occur even if the WSGI application itself is running in a daemon process.

A further example where this can be useful is where which daemon process is used is dependent on some attribute of the user. For example, if using the Apache configuration:

```
WSGIDaemonProcess django-admin
WSGIDaemonProcess django-users

WSGIProcessGroup %{ENV:PROCESS_GROUP}
```

which daemon process the request is delegated to can be controlled from the auth provider:

```
import apache.httpd

def check_password(environ, user, password):
    db.reset_queries()

    kwargs = {'username': user, 'is_active': True}

    try:
        try:
            user = User.objects.get(**kwargs)
        except User.DoesNotExist:
            return None

        if user.check_password(password):
            req = apache.httpd.request_rec(environ["apache.request_rec"])

            if user.is_staff:
                req.subprocess_env["PROCESS_GROUP"] = 'django-admin'
            else:
                req.subprocess_env["PROCESS_GROUP"] = 'django-users'

            return True
        else:
            return False
    finally:
        db.connection.close()
```

For more details see:

http://code.google.com/p/modwsgi/wiki/AccessControlMechanisms

4. When Apache 2.2 is being used, now possible to provide a script file containing a callable which returns the groups that a user is a member of. This can be used in conjunction with a 'group' option to the Apache 'Require' directive. Note that up to mod_wsgi 2.0c3 the option was actually 'wsgi-group'.

Apache configuration for defining an auth provider for Basic authentication and subsequent group authorisation would be:

```
AuthType Basic
AuthName "Top Secret"
AuthBasicProvider wsgi
WSGIAuthUserScript /usr/local/wsgi/scripts/auth.wsgi
WSGIAuthGroupScript /usr/local/wsgi/scripts/auth.wsgi
Require group secret-agents
```

```
Require valid-user
```

The 'auth.wsgi' script would then need to contain a 'check_password()' and 'groups_for_user()' function with a sample as shown below:

```
def check_password(environ, user, password):
    if user == 'spy':
        if password == 'secret':
            return True
        return False
    return None

def groups_for_user(environ, user):
    if user == 'spy':
        return ['secret-agents']
    return ['']
```

For more details see:

http://code.google.com/p/modwsgi/wiki/AccessControlMechanisms

5. Implemented WSGIDispatchScript directive. This directive can be used to designate a script file in which can be optionally defined any of the functions:

```
def process_group(environ):
    return "%{GLOBAL}"

def application_group(environ):
    return "%{GLOBAL}"

def callable_object(environ):
    return "application"
```

This allows for the process group, application group and callable object name for a WSGI application to be programmatically defined rather than be exclusively drawn from the configuration.

Each function if wishing to override the value defined by the configuration should return a string object. If None is returned then value defined by the configuration will still be used.

By default the script file code will be executed within the context of the '%{GLOBAL}' application group within the Apache child processes (never in the daemon processes). The application group used can be overridden by defining the 'application-group' option to the script directive. Note that up to 2.0c3 the WSGIServerGroup directive was instead provided, but this has now been removed.

This feature could be used as part of a mechanism for distributing requests across a number of daemon process groups, but always directing requests from a specific user to the same daemon process.

6. Implemented inactivity-timeout option for WSGIDaemonProcess directive. For example:

```
WSGIDaemonProcess trac processes=1 threads=15 \
  maximum-requests=1000 inactivity-timeout=300
```

When this option is used, the daemon process will be shutdown, and thence restarted, after no request activity for the defined period (in seconds).

The purpose of this option is to allow amount of memory being used by a process to be dropped back to the initial idle state level. This option would be used where the application delegated to the daemon process was used infrequently and thus it would be preferable to reclaim the memory when the application is not in use.

7. In daemon processes, the HOME environment variable is now overridden such that its initial value when a new Python sub interpreter is created is the same as the home directory of the user that the daemon process is running as. This is to give some certainty as to its value as otherwise the HOME environment variable may be that of the root user, a particular user, or the user that ran 'sudo' to start Apache. This is because HOME environment variable will be inherited from environment of user that Apache is started as and has no relationship to the user that the process is actually run as.

Note that the HOME environment variable is not updated for embedded mode as this would change the environment of code running under different Apache modules, such as mod_php and mod_perl. Not seen as being good practice to modify the environment of other systems.

Once consequence of the HOME environment variable being set correctly for daemon processes at least, is that the default location calculated for Python egg cache should then be correct. If running in embedded mode, would still be necessary to manually override Python egg cache location.

8. In daemon processes, the initial current working directory of the process will be set to the home directory of the user that the process runs as, or as specified by the 'home' option to the WSGIDaemonProcess directive.

9. Added 'stack-size' option to WSGIDaemonProcess so that per thread stack size can be overridden for processes in the daemon process group.

This can be required on Linux where the default stack size for threads is the same as the default user process stack size, that being 8MB. When running in a VPS provided by a web hosting company, where they for some reason seem to take into consideration the virtual memory size as well as the resident memory size when calculating your process limits, it is better to drop the per thread stack size down to a value closer to 512KB. For example:

```
WSGIDaemonProcess example processes=2 threads=25 stack-size=524288
```

10. Added some direct support into mod_wsgi for virtual environments for Python such as virtualenv and workingenv.

The first approach to configuration is to use WSGIPythonPath directive at global scope in apache configuration. For example:

```
# workingenv
WSGIPythonPath /some/path/env/lib/python2.3

# virtualenv
WSGIPythonPath /some/path/env/lib/python2.3/site-packages
```

The path you have to specify is slightly different depending on whether you use workingenv or virtualenv packages.

Previously the WSGIPythonPath directive would just override the `PYTHONPATH` environment variable. Instead it now calls `site.addsitedir()` for any specified directories, thus triggering the reading of any .pth files and the subsequent addition of further directories there specified to sys.path.

Note that directories added with WSGIPythonPath only apply to applications running in embedded mode.

If you want to specify directories for daemon processes, you can use the 'python-path' option to WSGIDaemonProcess. For example:

```
WSGIDaemonProcess turbogears processes=5 threads=1 \
  user=site1 group=site1 maximum-requests=1000 \
  python-path=/some/path/env/lib/python2.3/site-packages

WSGIScriptAlias / /some/path/scripts/turbogears.wsgi

WSGIProcessGroup turbogears
WSGIApplicationGroup %{GLOBAL}
WSGIReloadMechanism Process
```

Do note that anything defined in the standard Python site-packages directories takes precedence over directories added using the mechanisms described above. Thus, if wanting to use these virtual environments all the time, your standard Python installation effectively needs to have an empty site-packages directory. Alternatively, on UNIX systems you can use the WSGIPythonHome directive to point to a virtual environment which contains an empty 'site-packages'.

End result is that with these options, should be very easy to have different daemon process groups using different Python virtual environments without any fiddles having to be done in the WSGI script file itself.

For more details see:

http://code.google.com/p/modwsgi/wiki/VirtualEnvironments

11. Added WSGIPythonEggs directive and corresponding 'python-eggs' option for WSGIDaemonProcess directive. These allow the location of the Python egg cache directive to be set for applications running in embedded mode or in the designated daemon processes. These options have the same affect as if the 'PYTHON_EGG_CACHE' environment variable had been set.

12. Implement 'deadlock-timeout' option for WSGIDaemonProcess for detecting Python programs that hold the GIL for extended periods, thus perhaps indicating that process has frozen or has become unresponsive. The default value for the timeout is 300 seconds.

13. Added support for providing an access control script. This equates to the access handler phase of Apache and would be use to deny access to a subset of URLs based on the details of the remote client. The path to the script is defined using the WSGIAccessScript directive:

```
WSGIAccessScript /usr/local/wsgi/script/access.wsgi
```

The name of the function that must exist in the script file is 'allow_access()'. It must return True or False:

```
def allow_access(environ, host):
    return host in ['localhost', '::1']
```

This function will always be executed in the context of the Apache child processes even if it is controlling access to a WSGI application which has been delegated to a daemon process. By default the function will be executed in the context of the main Python interpreter, ie., '%{GLOBAL}'. This can be overridden by using the 'application-group' option to the WSGIAccessScript directive:

```
WSGIAccessScript /usr/local/wsgi/script/access.wsgi application-group=admin
```

For more details see documentation on [AccessControlMechanisms Access Control Mechanisms]

14. Added support for loading a script file at the time that process is first started. This would allow modules related to an application to be preloaded into an interpreter immediately rather than it only occuring when the first request arrives for that application.

The directive for designating the script to load is WSGIImportScript. The directive can only be used at global scope within the Apache configuration. It is necessary to designate both the application group, and if dameon mode support is available, the process group:

```
WSGIImportScript /usr/local/wsgi/script/import.wsgi \
 process-group=%{GLOBAL} application-group=django
```

14. Add "–disable-embedded" option to "configure" script so that ability to run a WSGI application in embedded mode can be disabled completely. Also added the directive WSGIRestrictEmbedded so that ability to run a WSGI application in embedded mode can be disabled easily if support for embedde mode is still compiled in.

15. Added support for optional WSGI extension wsgi.file_wrapper. On UNIX systems and when Apache 2.X is being used, if the wrapped file like object relates to a regular file then additional optimisations will be applied to improve the performance of returning the file in a response.

16. Added 'display-name' option for WSGIDaemonProcess. On operating systems where it works, this should allow displayed name of daemon process shown by 'ps' to be changed. Note that name will be truncated to whatever the existing length of 'argv[0]' was for the process.

17. When WSGI application generates more content than what was defined by response content length header, excess is discarded. If Apache log level is set to debug, messages will be logged to Apache error log file warning of when generated content length differs to specified content length.

18. Allow WSGIPassAuthorization to be used in .htaccess file if !FileInfo override has been set. This has been allowed as !FileInfo enables ability to use both mod_rewrite and mod_headers, which both provide means of getting at the authorisation header anyway, so no point trying to block it.

19. Optimise sending of WSGI environment across to daemon process by reducing number of writes to socket. For daemon mode and a simple hello world application this improves base performance by 40% moving it significantly closer to performance of embedded mode.

20. Always change a HEAD request into a GET request. This is to ensure that a WSGI application always generates response content. If this isn't done then any Apache output filters will not get to see the response content and if they need to see the response content to generate headers based on it, then the response headers from a HEAD request would be incorrect and not match a GET request as required.

If Apache 2.X, this will not however be done if there are no Apache output filters registered which could change the response headers or content.

21. Add option "send-buffer-size" and "receive-buffer-size" to WSGIDaemonProcess for controlling the send and receive buffer sizes of the UNIX socket used to communicate with mod_wsgi daemon processes. This is to work around or limit deadlock problems that can occur in certain cases when the operating system defines a very small default UNIX socket buffer size.

22. When no request content has been read and headers are to be sent back, force a zero length read in order to flush out any '100 Continue' response if expected by client. This is only done for 2xx and 3xx response status values.

23. A negative value for content length in response wasn't being rejected. Where invalid header was being returned in response original response status was being returned instead of a 500 error.

Version 1.6

Version 1.6 of mod_wsgi can be obtained from:

> http://modwsgi.googlecode.com/files/mod_wsgi-1.6.tar.gz

Note that this is a quick followup to version 1.5 of mod_wsgi to rectify significant problem introduced by that release. You should therefore also refer to:

- *Version 1.5*.

Bug Fixes

1. Fixed problem introduced in version 1.5 of mod_wsgi whereby use of daemon mode would cause CGI scripts to fail.

It is quite possible that the bug could also have caused failures with other Apache modules that relied on registering of cleanup functions against Apache configuration memory pool.

For details see:

> http://groups.google.com/group/modwsgi/browse_frm/thread/79a86f8faffe7dcf

Version 1.5

Version 1.5 of mod_wsgi can be obtained from:

http://modwsgi.googlecode.com/files/mod_wsgi-1.5.tar.gz

Bug Fixes

1. Fix bug where listener socket file descriptors for daemon processes were being leaked in Apache parent process on a graceful restart. Also fixes problem where UNIX listener socket was left in filesystem on both graceful restart and graceful shutdown. For details see:

http://code.google.com/p/modwsgi/issues/detail?id=95

This is a backport of change from version 2.2 of mod_wsgi.

Version 1.4

Version 1.4 of mod_wsgi can be obtained from:

http://modwsgi.googlecode.com/files/mod_wsgi-1.4.tar.gz

Bug Fixes

1. A negative value for content length in response wasn't being rejected. Where invalid header was being returned in response original response status was being returned instead of a 500 error.

2. Fix bug which was resulting in logging destined for !VirtualHost !ErrorLog going missing or ending up in main Apache error log.

http://code.google.com/p/modwsgi/issues/detail?id=79

Features Added

1. Optimise sending of WSGI environment across to daemon process by reducing number of writes to socket. For daemon mode and a simple hello world application this improves base performance by 40% moving it significantly closer to performance of embedded mode.

This is a backport of change from version 2.0 of mod_wsgi.

Version 1.3

Version 1.3 of mod_wsgi can be obtained from:

http://modwsgi.googlecode.com/files/mod_wsgi-1.3.tar.gz

Bug Fixes

1. Fix bug whereby mod_wsgi daemon process could hang when a request with content greater than UNIX socket buffer size, was directed at a WSGI application resource handler which in turn returned a response, greater than UNIX socket buffer size, without first consuming the request content.

There were two aspects to this problem, the first is that the above would trigger that specific request to hang. Second was that at the point of the hang, the Python GIL hadn't been released, and so all other threads were blocked from running any Python code resulting in whole process effectively hanging.

Code now correctly ensures that Python GIL is released prior to going into potentially blocking operation. Secondly, where mutual deadlock between Apache child process and mod_wsgi daemon process, timeout as defined by the standard Apache 'Timeout' directive will now kick in and remaining request content discarded by Apache child process so that thread in the daemon process can continue and break out of its hung state.

Although this can still result in request thread being in a hung state until the timeout occurs, this mirrors exactly what would happen if running a WSGI application using a CGI-WSGI bridge behind Apache mod_cgi module. A better solution which would avoid the hung state altogether is still being investigated.

Note that this scenario shouldn't ever eventuate for a correctly implemented and functioning web application, however it is feasible that it could be triggered as a result of spambots which attempt to POST data randomly to sites with the hope they find a wiki system with an unprotected comment system.

Version 1.2

Version 1.2 of mod_wsgi can be obtained from:

> http://modwsgi.googlecode.com/files/mod_wsgi-1.2.tar.gz

Bug Fixes

1. When headers are flushed by mod_wsgi is not strictly compliant with the WSGI specification. In particular the specification says:

> The start_response callable must not actually transmit the response headers. Instead, it must store them for the server or gateway to transmit only after the first iteration of the application return value that yields a non-empty string, or upon the application's first invocation of the write() callable. In other words, response headers must not be sent until there is actual body data available, or until the application's returned iterable is exhausted. (The only possible exception to this rule is if the response headers explicitly include a Content-Length of zero.)

In mod_wsgi when an iterable was returned from the application, the headers were being flushed even if the string was empty. See:

> http://code.google.com/p/modwsgi/issues/detail?id=35

2. Calling start_response() a second time to supply exception information and status to replace prior response headers and status, was resulting in a process crash when there had actually been response content sent and the existing response headers and status flushed and written back to the client. See:

> http://code.google.com/p/modwsgi/issues/detail?id=36

3. Added additional logging to highlight instance where WSGI script file was removed in between the time that Apache matched request to it and the WSGI script file was loaded and the request passed to it. These changes also log something if the attempt to stat the WSGI script file in the daemon process fails due to inadequate permissions or other reasons.

4. Fixed a few instances where logging via request object before fake request object in daemon process had been constructed properly. The particular cases would only have been triggered if something other than mod_wsgi code with Apache child process had tried to communicate with the daemon process.

5. Fixed problem when Apache 1.3 or 2.0 was being used, where the automatically determined default for the application group (interpreter) name would be wrong where the URL had repeating slashes in it after the leading portion of the URL which mapped to the mount point of the WSGI application. See:

http://code.google.com/p/modwsgi/issues/detail?id=39

In particular, for a URL with the repeating slash the application group name would have a trailing slash appended when it shouldn't. The consequences of this are that two instances of the WSGI application could end up being loaded into the same process, doubling the memory usage for the process.

Besides the additional memory use, this would in general not be an issue as most applications would be designed to work within multi process environment of Apache. If however a specific application was designed to only work within a single process (interpreter instance), as would occur when Windows was being used, or a single daemon process with daemon mode, then there may be issues as requests which had a repeating slash in the URL would not access the same application data as those without.

Note, this problem could only arise where WSGIApplicationGroup directive wasn't used and thus default value being used. Or the value '%{RESOURCE}' was specified as argument to WSGIApplicationGroup, this being the same as the default.

6. Fixed problem whereby status of sub processes created from mod_wsgi daemon processes were not being caught properly. This was because mod_wsgi was wrongly blocking SIGCHLD signal. See:

http://code.google.com/p/modwsgi/issues/detail?id=38

Version 1.1

Version 1.1 of mod_wsgi can be obtained from:

http://modwsgi.googlecode.com/files/mod_wsgi-1.1.tar.gz

Bug Fixes

1. Fix bug which could result in processes crashing when multiple threads attempt to write to sys.stderr or sys.stdout at the same time. See:

http://code.google.com/p/modwsgi/issues/detail?id=30

Chance of this occuring was small, as was contingent on code writing out strings which contained an embedded newline but no terminating new line, thereby triggering the internal line caching code.

2. In error case when not able to release interpreter, was wrongly trying to release Python GIL around code to unlock module mutex when didn't actually have the GIL acquired in the first place. Didn't strictly need to be releasing GIL when releasing lock as it shouldn't block anyway, so don't do this even in case where had the Python GIL.

This problem would only have been encountered in situation where Python had failed in a major way to begin with.

3. Incorrectly trying to output Python exception details when Python GIL would not have been held.

This problem would only have been encountered in situation where Python had failed in a major way to begin with.

4. Fix location of Python object reference count decrements to avoid decrement reference count on null pointer.

Would only have caused a problem if Python was in some sort of corrupted state to begin with as the object which the reference count was being performed on should always exist.

5. Replace normal Apache connection setup in daemon processes with equivalent code that avoids possibility that other Apache modules will insert their own connection level input/output filters. This is needed as running WSGI applications in daemon processes where requests were arriving to Apache as HTTPS requests could cause daemon processes to crash. See:

http://code.google.com/p/modwsgi/issues/detail?id=33

This was only occuring for some HTTPS configurations, but not known what exactly was different about those configurations to cause the problem. Actually possible that the real problem was mod_logio as described below.

6. Substitute optional ap_logio_add_bytes_out() function provided by the mod_logio module when loaded and when handling request in daemon process. This is needed to prevent core output filters calling this function and triggering a crash due to configuration for mod_logio not being setup. See:

http://code.google.com/p/modwsgi/issues/detail?id=34

Version 1.0

Version 1.0 of mod_wsgi can be obtained from:

http://modwsgi.googlecode.com/files/mod_wsgi-1.0.tar.gz