mod_wsgi Learning

Friday, April 04, 2014 14:46

http://modwsgi.readthedocs.org/en/latest/configuration-guides/assorted-guides.html#defining-process-groups
http://modwsgi.readthedocs.org/en/latest/developer-guides/processes-and-threading.html

1. **Official Website:** http://modwsgi.readthedocs.org/en/latest/

2. **What Is mod_wsgi?**
The mod_wsgi package implements a simple to use Apache module which can host any Python application which supports the Python WSGI interface. The module is suitable for use in hosting high performance production web sites, as well as your average self managed personal sites running on a virtual private server or shared host.

3. **Modes Of Operation**
When hosting WSGI applications using mod_wsgi, one of two primary modes of operation can be used:
   a. **Embeded mode**: mod_wsgi works in a similar way to mod_python in that the Python application code will be executed within the context of the normal Apache child processes. WSGI applications when run in this mode will therefore share the same processes as other Apache hosted applications using Apache modules for PHP and Perl.
   b. **Daemon mode**: This mode operates in similar ways to FASTCGI/SCGI solutions, whereby distinct processes can be dedicated to run a WSGI application. Unlike FASTCGI/SCGI solutions however, neither a separate process supervisor or WSGI adapter is needed when implementing the WSGI application and everything is handled automatically by mod_wsgi.

4. **Server Performance**
The mod_wsgi module is written in C code directly against the internal Apache and Python application programming interfaces. As such, for hosting WSGI applications in conjunction with Apache it has a lower memory overhead and performs better than existing WSGI adapters for mod_python or alternative FASTCGI/SCGI/CGI or proxy based solutions.

Although embedded mode can technically perform better, daemon mode would generally be the safest choice to use. This is because to get the best performance out of embedded mode you must tune the Apache MPM settings, which in their default settings are biased towards serving of static media and hosting of PHP applications. If the Apache MPM settings are not set appropriately for the type of application being hosted, then worse performance can be seen rather than better performance.

Thus, unless you are adept at configuring Apache, always use daemon mode when available. Overall, for large Python web applications you wouldn't normally expect to see any significant difference between daemon mode and embedded mode, as the bottlenecks are going to be in the Python web application or any database access.

5. **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 = '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:eezw

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

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'.

6. **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.

7. **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 [ConfigurationDirectives Configuration Directives].

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.

8. **Defining Process Groups**

```
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>
```

For further information about the options that can be supplied to the WSGIDaemonProcess directive see the more detailed documentation on the [ConfigurationDirectives Configuration Directives]. 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 multiprocess 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.

9. **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.

10. **Configuration Directives**
    - WSGIAcceptMutex
    - WSGIAccessScript
    - WSGIApplicationGroup
    - WSGIAuthGroupScript
    - WSGIAuthUserScript
    - WSGICallableObject
    - WSGICaseSensitivity
    - WSGIDaemonProcess
    - WSGIImportScript
    - WSGILazyInitialization
    - WSGIPassAuthorization
    - WSGIProcessGroup
    - WSGIPythonEggs
    - WSGIPythonHome
    - WSGIPythonOptimize
    - WSGIPythonPath
    - WSGIRestrictEmbedded
    - WSGIRestrictProcess
    - WSGIRestrictSignal
    - WSGIRestrictStdin
    - WSGIRestrictStdout
    - WSGIScriptAlias
    - WSGIScriptAliasMatch
    - WSGIScriptReloading
    - WSGISocketPrefix

11.
12. **Running A Basic Application**