WSGI - Gateway or Glue?

Author: Mark Rees

Abstract

The Web Service Gateway Interface (WSGI) is an API defined in PEP-0333 that allows communication between web servers and Python web applications. At the time of OSDC 2005, the PEP [pep] will be 2 years old and the paper will review the state of WSGI implementations for the various web servers including Apache, IIS, Twisted etc. It will also show how the API is used to implement a web server interface and create an application. In the last 6 months, the main WSGI community focus has been on creating web applications by composing them from components that communicate via WSGI. The remainder of the paper will discuss this use of WSGI middleware components.

Introduction

WSGI is the brainchild of Phillip J. Eby [pje] who was looking for a simple and universal interface between web servers and python web applications or frameworks. Originally called the Python Web Container Interface, the pre-PEP was posted to the Python Web-SIG on December 8 2003. After robust discussion and input from many of the Python community it became the Web Server Gateway Interface and was submitted as Python Enhancement Proposal 333 [pep].

So what problems does WSGI attempt to solve? A Python programmer has many web frameworks to choose from and normally as part of a framework selection process, what webservers the framework runs on can be a deciding factor. For a framework author, finding the time to create adaptors for all webservers is an impossible task. If the framework author programmes to the WSGI Application Gateway API, then the user of the framework can pick what ever webserver they want as long as the webserver has a WSGI adaptor.

Since the components of any web framework designed to work with WSGI use the same interface, there is the potential to use components from multiple web frameworks in a single application.

How WSGI works

The PEP [pep] is well written, very understandable, and is the ultimate source of WSGI knowledge. But here is a basic overview of how WSGI works.

WSGI specifies two interfaces, an interface for the webserver to communicate with the application, and an interface for the application to communicate with the webserver.

For the webserver to communicate with the application, it calls a function or callable object that the application supplies. This function or object accepts two positional arguments, environ and start_response. The environ argument must be a builtin Python dictionary containing CGI-style environment variables like REQUEST_METHOD [cgi],required WSGI variables, and may also include server-specific extension variables. The start_response argument will be a Python function.

For the application to communicate with the webserver, the application prepares the headers it wants to send, then calls the start_response function that it was given, with a status code and the list of headers. The application then prepares the body of the response as a list of strings or an iterator. The response body is passed back to the webserver by returning it.

On receipt of the list or iterator from the application, the webserver streams the strings to the application client.

So show me the code. Consider the simplistic CGI script code below:

```
#!/usr/bin/env python
print 'Content-type: text/plain\n\n'
print 'Hello world!'
```

As a WSGI application we could use this code:

```
def application(environ, start_response):
    """Simplest possible application object"""
    status = '200 OK'
    response_headers = [('Content-type','text/plain')]
    start_response(status, response_headers)
    return ['Hello world!\n']
```

or more OO like:

```
class WSGIAppClass:
    """Produce the same output, but using a class
    """

    def __init__(self, environ, start_response):
        self.environ = environ
        self.start = start_response

    def __iter__(self):
        status = '200 OK'
        response_headers = [('Content-type','text/plain')]
        self.start(status, response_headers)
        yield "Hello world!\n"
```

Now to test my code I could configure the Simple HTTP Server from wsgiref [wsgiref] to run it:

```
from wsgiref import simple_server
import hello_world

httpd = simple_server.WSGIServer(('',8000),simple_server.WSGIRequestHandler)
httpd.set_app(hello_world.application)
httpd.serve_forever()
```

Then I could deploy on Apache using FastCGI:

```
from flup.server.fcgi import WSGIServer
import hello_world

WSGIServer(hello_world.application).run()
```

Or deploy on IIS using ISAPI:

```
import isapi_wsgi
import hello_world
# The entry points for the ISAPI extension.
def __ExtensionFactory__():
    return isapi_wsgi.ISAPISimpleHandler(hello = hello_world.application)

if __name__=='__main__':
    # If run from the command-line, install ourselves.
    from isapi.install import *
    params = ISAPIParameters()
    # Setup the virtual directories - this is a list of directories our
    # extension uses - in this case only 1.
    # Each extension has a "script map" - this is the mapping of ISAPI
    # extensions.
    sm = [
        ScriptMapParams(Extension="*", Flags=0)
    ]
```

```
    vd = VirtualDirParameters(Name="isapi-wsgi-hello",
                              Description = "ISAPI-WSGI Hello WOrld",
                              ScriptMaps = sm,
                              ScriptMapUpdate = "replace"
                              )
    params.VirtualDirs = [vd]
    HandleCommandLine(params)
```

With the exception of the ISAPI deployment, deploying under most other WSGI webserver adaptors involves very simple scripts.

To gain a better understanding of how a webserver adaptor is implemented, the CGI adaptor described in PEP_0333 [pep] is a great starting point.

WSGI Web Server Implementations

As the table below shows, due to the efforts of a few it is possible to deploy a WSGI application on the most common webserver runtime environments. The implementations listed in the table only covers the adaptors the author has evaluated and if you cannot see an environment you need to run under, then it is possible someone is working on it.

Implementation	Description
wsgiref	Written by the PEP-0333 author, this package includes a WSGI SimpleHTTPServer but more importantly provides a library of base classes and utilities that are useful for application, framework and WSGI server implementations. [wsgiref]
modjy	A Jython WSGI implementation that runs in a servlet engine. [modjy]
WSGIUtils	A multi-threaded WSGI server implementation. [wsgiutils]
Twisted	Currently 2 adaptors for Twisted. The original developed by Peter Hunt [twisted_wsgi] and the WSGI module in twisted.web2 [twisted_wsgi2]
isapi_wsgi	A WSGI adaptor for ISAPI under IIS. Single threaded with multi-threaded version under development. [isapi_wsgi] This adaptor was written by the author.
IIS/ASP	A WSGI adaptor for IIS using ASP. Supports multiple threads. [asp_wsgi]
flup	Three sets of WSGI servers/gateways, which speak AJP 1.3, FastCGI, and SCGI. Each server comes in two flavours: a threaded version, and a forking version. [flup]
mod_python	A WSGI adaptor for mod_python. [mod_python]
SWAP	WSGI server for SCGI [swap]
PEAK	PEAK now offers three options for running WSGI applications: CGI, FastCGI, and SimpleHTTPServer. [peak]

WSGI Enabled Frameworks

Unless you are writing a web framework or WSGI capable replacement for CGI scripts, the most likely exposure to WSGI will be using a web framework that supports WSGI. So if you decide to use any of the WSGI enabled web frameworks, then you should be able to deploy them via the WSGI webserver implementations listed in the previous section. The table below is not a complete list of the Python web frameworks that now have WSGI support, but should give you an indication of the takeup of WSGI by framework developers to simplify deployment.

Framework	Description

CherryPy 2.1	CherryPy [cherrypy] is a pythonic, object-oriented web development framework with full WSGI support and also bundles a multi-threaded WSGI server. Since TurboGears [turbogears] and Subway [subway] use CherryPy as their webserver gateway, they have WSGI support by default.
Paste Webkit	Paste Webkit [webkit] is a reimplementation of Webware API but uses PythonPaste [paste] WSGI middleware components.
Python Web Modules	A set of Python web modules [pywebmod] to allow development of CGI or WSGI web applications. Also provides a number WSGI middleware components
Django	Django is a high-level Python Web framework that encourages rapid development [django]

WSGI Middleware

Over the last 6 months, much of the WSGI discussion and development has been focused on WSGI Middleware. That is, using python components that support both the WSGI application and server API's and pipelining these components together to create a web application. So a simple WSGI Authentication component that we could wrap around an existing WSGI application could be written as follows:

```
class AuthenticationMiddleware:
    def __init__(self, app, allowed_usernames):
        self.app = app
        self.allowed_usernames = allowed_usernames

    def __call__(self, environ, start_response):
        if environ.get('REMOTE_USER') in self.allowed_usernames:
            return self.app(environ, start_response)
        start_response(
            '403 Forbidden', [('Content-type', 'text/html')])
        return ['You are forbidden to view this resource']
```

Now to deploy my code I could use a WSGI CGI adaptor based on the example in PEP-0333 [pep] to run it as a CGI process:

```
#!/usr/local/bin/python2.4
import hello_world
import middleware

allowed_users = ['guido','monty']

if __name__ == '__main__':
    from cgi_wsgi import run_with_cgi
    run_with_cgi(middleware.AuthenticationMiddleware(
        hello_world.application,allowed_users))
```

The WSGIUtils [wsgiutils], Python Paste [paste], flup [flup] and Python Web Modules [pywebmod] packages provide middleware components that cover session management, error handling, authentication, compression and URL parsing. Another excellent example of WSGI middleware is Leslie Orchards [wsgi_xslt] component for interpreting XSLT. This means any WSGI application can output XML and have it transformed using an XSLT stylesheet.

For an example of a complete application using middleware have a look at PyFileServer [pyfileserver], a WSGI web application for sharing filesystem directories over WebDAV.

Configuration

Being able to use a framework-neutral set of components to create the "ultimate" Python web application is not without it's problems. The code to combine all the middleware components can get very messy. If you

consider the following pseudocode example:

```
def configure(app):
    return ErrorHandlerMiddleware(
            SessionMiddleware(
             IdentificationMiddleware(
              AuthenticationMiddleware(
               UrlParserMiddleware(app))))))

if __name__ == '__main__':
    app = Application()
    app_pipeline = configure(app)
    server = Server(app_pipeline)
    server.serve()
```

Once you add all the arguments that need to be passed, code maintenance and debugging is likely to become a nightmare. Developing a standardised way of configuring WSGI applications has been a well discussed topic on the Python Web-SIG [websig]. Ian Bicking has released Paste Deployment [paste.deploy] which is a system for finding and configuring WSGI applications and servers. To use Paste deployment to configure and load an WSGI application requires, adding an app factory wrapper to your WSGI application:

```
def application(environ, start_response):
    """Simplest possible application object"""
    status = '200 OK'
    response_headers = [('Content-type','text/plain')]
    start_response(status, response_headers)
    return ['Hello world!\n']

def app_factory(global_config, **local_config):
    return application
```

a simple configuration file:

```
[app:main]
paste.app_factory = hello_world:app_factory
```

and code similar to:

```
from paste.deploy import loadapp
wsgi_app = loadapp('config:/path/to/config.ini')
```

And the WSGI deployment code for the runtime environment, in this case FastCGI:

```
from flup.server.fcgi import WSGIServer
WSGIServer(wsgi_app).run()
```

The Paste Deployment configuration file has directives for defining multiple applications, global and local configuration variables, filters and defining pipelines of the filters. This pipeline definition feature should simplify the creation of applications with WSGI middleware. For the details of how to setup a configuration, the Paste Deployment website [paste.deploy] has good documentation.

Conclusions

So what is WSGI, a gateway or glue? I hope this paper has shown that it is both. A well defined API for communication between webservers and python web applications, and that it can be used to "glue" together a framework-neutral set of components. Until more applications are developed using the WSGI gateways and frameworks discussed in the paper, we will not discover if there are issues to be resolved. But I believe the gateways and frameworks are now in a state where production applications can be developed and if issues are discovered, the community will work towards resolving them asap.

References

[pje] Phillip J. Eby: Author of WSGI PEP-0333 (http://dirtsimple.org/)

[peak] PEAK: The Python Enterprise Application Kit (http://peak.telecommunity.com/)

[pep] PEP-0333: Python Web Server Gateway Interface v1.0 (http://www.python.org/peps/pep-0333.html)

[cgi] The Common Gateway Interface Specification (http://hoohoo.ncsa.uiuc.edu/cgi/interface.html)

[wsgiref] WSGI Reference Package (http://svn.eby-sarna.com/wsgiref/)

[modjy] Jython WSGI adaptor (http://www.xhaus.com/modjy/)

[wsgiutils] WSGIUtils (http://www.owlfish.com/software/wsgiutils/)

[twisted_wsgi] WSGI adaptor for Twisted (http://svn.pythonpaste.org/Paste/tags/WSGIKit/wsgikit/twisted_wsgi.py)

[twisted_wsgi2] WSGI module for Twisted Web2 (http://twistedmatrix.com/documents/current/api/twisted.web2.wsgi.html)

[isapi_wsgi] WSGI adaptor of ISAPI (http://isapi-wsgi.python-hosting.com/)

[asp_wsgi] WSGI adaptor for ASP (http://www.aminus.org/blogs/index.php/fumanchu/2005/05/26/wsgi_gateway_for_asp_microsoft_iis)

[flup] flup WSGI packages (http://www.saddi.com/software/flup/)

[mod_python] WSGI adaptor for mod_python (http://www.aminus.org/blogs/index.php/fumanchu/2005/05/17/wsgi_wrapper_for_mod_python)

[swap] WSGI adaptor for SCGI (http://svn.pythonpaste.org/Paste/tags/WSGIKit/wsgikit/scgiserver.py)

[cherrypy] CherryPy web framework (http://www.cherrypy.org)

[turbogears] TurboGear web framework (http://www.turbogears.org)

[subway] Subway web framework (http://subway.python-hosting.com)

[webkit] WebKit web framework (http://svn.pythonpaste.org/Paste/WebKit)

[paste] Python component based web development tools (http://www.pythonpaste.org)

[pywebmod] Python Web Modules (http://www.pythonweb.org/)

[django] Django web framework (http://www.djangoproject.com/)

[wsgi_xslt] Leslie Orchard, Discovering WSGI and XSLT as Middleware (http://www.decafbad.com/blog/2005/07/18/discovering_wsgi_and_xslt_as_middleware)

[websig] Threads discussing WSGI configuration (http://mail.python.org/pipermail/web-sig/2005-July/001511.html) (http://mail.python.org/pipermail/web-sig/2005-July/001512.html) (http://mail.python.org/pipermail/web-sig/2005-July/001533.html)

[paste.deploy] Paste Deployment (http://pythonpaste.org/deploy/)

[pyfileserver] WSGI WebDAV File Server (http://pyfilesync.berlios.de/pyfileserver.html)