PYTHON WEB APPLICATIONS AND WSGI

Abduaziz Ziyodov
Student Inha University in Tashkent
abduaziz.ziyodov@mail.ru
https://doi.org/10.5281/zenodo.8205282

Abstract

This article specifies an interface between web servers and Python web applications. In the Python ecosystem this is crucial because, without WSGI, python web applications will not be able to work with our web servers. The author explains a detailed study of WSGI with its applications, goals, implementation and specification details.

Keywords

Python, WSGI, Web Server Gateway Interface, Web, CGI, PEP, Flask, Django, HTTP

Introduction

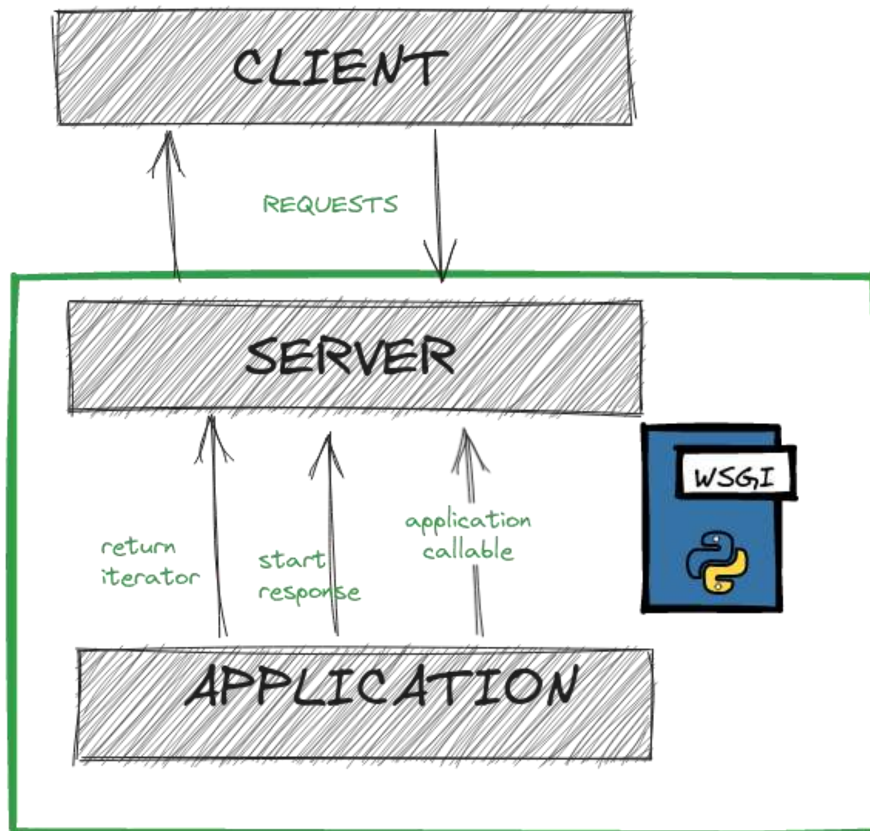
In the 1990s there were millions of websites, and pages were static. However, they needed to produce dynamic content. Then Apache developed a new solution: mod_python HTTP server module that integrates the python with the web server. But it was not an effective solution, there are some critical disadvantages of mod_python:

- It is no longer actively developed or maintained and there have been no security updates.
- It is really slow, mod_python embeds the python into the Apache server, which can slow down performance.
- Apache mod_python creates a new process for each new request which costs very expensive for hardware.

The real problem was the lack of a universal interface/specification for developing web applications by Python code.

WSGI

On December 7, 2003, Python Enhancement Proposal (**PEP-333**) was published by Phillip J. The core idea was to provide a high-level, universal interface between Python applications and web servers. This document clearly describes the goals of this proposal, applicPythonframework and server/gateway side, implementations and many-many more. As I mentioned earlier, in 1990 developers had to create only static web applications. Such kinds of applications just needed to send HTML documents (static content) to the client's browser. By using the built-in HTTP library's BaseHTTPHandler, developers could achieve such results. However, nowadays we need to server/manipulate dynamic content. That's why, we need WSGI. Web Server Gateway Interface (WSGI) is a standard that facilitates interaction between the server and the application by acting as an interface. It is currently the most preferred interface for Python web programming.



How WSGI works

There are two major sides to the WSGI framework:

- You need to have a WSGI application - that is Python callable object.
- Server gateways like Apache or Nginx as I mentioned above.

Functions in Python are callable objects by default, but If you want to create a callable object from the class you need to implement __call__ method:

```
class Application:
    def __call__(self, environ, start_response):
        ...
```

It receives a request from the HTTP clients redirected to our application, and an application callable is invoked by the gateway. Our WSGI application accepts two positional arguments: environ which is a simple Python dictionary, and start_response callback function which will be used to send responses to the clients. The environment contains information about the request method, path info, query string, content type and some information about the host that the application is serving. The WSGI application returns an iterator, which will be iterated and according to its values responses will be sent to the client:

```
for data in Application(environ, start_response):
    # send data to a client
```

Currently, almost all modern Python web frameworks such as Django, flask are based on WSGI. To implement a working example of a WSGI application we need to know the fundamental things:

- environ - simply that is what we received from the client
- start_response - just a callback function, which takes HTTP status and headers.

We can get all the required information(request body, method, content type, path, etc.) from environ, and we just need to process all of them. Yes, that was the purpose of WSGI - Flexibility.

WSGI Application, Example

```
def application (environ,start_response):
  response_body = Method: %s' % environ['REQUEST_METHOD']
  status = '200 OK'
  response_headers = [
    ('Content-Type', 'text/plain'),
    ('Content-Length', str(len(response_body)))
  ]
  start_response(status, response_headers)
  return [response_body]
```

We are getting the request method from environ, creating an “OK” HTTP status code and generating the required HTTP headers. At the end function returns response_body as an iterable object in our case it is just a Python list. However, if you run this script nothing happens. Why? Because it lacks the server instantiation step:

```
from wsgiref.simple_server import make_server

... # application source code
httpd = make_server (
  '0.0.0.0',
  8080,
  application
)

if __name__ == "__main__":
  httpd.handle_request()
```

wsgiref is Python's bundled WSGI server, it takes to host, port and WSGI application as an argument. Then creates a WSGI server instance, that can be run. If we run this script, it only responds to one request. But if we use httpd.server_forever() instead of httpd.handle_request() we could achieve our goal.

In the next steps, you can content-type to text/html and then you can return HTML documents. If you manipulate your HTML documents with Python code, you can create dynamic web applications! That was the purpose of WSGI.

Modern Python Web Frameworks

Modern Python web frameworks are fully based on WSGI. They have many extra high-level components. Developers that use modern frameworks don't have to know about the environs. The reason is web frameworks (flask, Django) has specific parsers that convert WSGI environs to "Request" objects. Instead of using the start_response function, the developer needs to use the "Response" object which will be invoked by the start_response function in the lower level.

Modern frameworks have url dispatchers, and routing modules and frameworks like Django-rest-framework have serializer classes that parse and validate request content according to schema. That is incredible. Nowadays, in production mode wsgiref is not recommended. Python ecosystem has WSGI servers like Gunicorn, cherrypy and uWSGI.

Conclusion

WSGI is only part of the solution. It helps interoperability but does nothing for the Python newbie who just wants to get a simple dynamic site up and running quickly and is overwhelmed by the choices. By knowing this concept, python developers can understand how Python web applications work under the hood. WSGI will handle requests synchronously, but there is a new implementation of this gateway and it is called ASGI.

References:

https://peps.python.org/pep-0333/
https://wsgi.readthedocs.io/en/latest/learn.html
https://linuxgazette.net/115/orr.html
https://link.springer.com/chapter/10.1007/978-1-4302-0534-0_16
https://www.researchgate.net/publication/352867984_A_Python-WSGI_and_PHP-Apache_Web_Server_Performance_Analysis_by_Search_Page_Generator_SPG