WSGI Tutorial Notes

Tuesday, April 22, 2014 16:55

1. Overview

What WSGI is not: a server, a python module, a framework, an API or any kind of software.
What it is: an interface specification by which server and application communicate. Both server and application interface sides are specified. It does not exist anywhere else other than as words in the PEP 3333. If an application (or framework or toolkit) is written to the WSGI spec then it will run on any server written to that spec.

WSGI applications (meaning WSGI compliant) can be stacked. Those in the middle of the stack are called middleware and must implement both sides of the WSGI interface, application and server. For the application in top of it it will behave as a server and for the application (or server) bellow as an application.

A WSGI server (meaning WSGI compliant) only receives the request from the client, pass it to the application and then send the response provided by the application to the client. It does nothing else. All the gory details must be supplied by the application or middleware.

It is not necessary to learn the WSGI spec to use frameworks or toolkits. To use middleware one must have a minimum understanding of how to stack them with the application or framework unless it is already integrated in the framework or the framework provides some kind of wrapper to integrate those that are not.

Python 2.5 and later comes with a WSGI server which will be used in this tutorial. In 2.4 and earlier it can be installed. For anything other than learning I strongly recommend Apache with mod_wsgi.

All the code in this tutorial is low level and has the sole purpose to be didactic by showing the WSGI specification at work. It is not meant for real use. For production code use toolkits, frameworks and middleware.

2. Application Interface

The WSGI application interface is implemented as a callable object: a function, a method, a class or an instance with a __call__ method.
That callable

- must accept two positional parameters:
  - A dictionary containing CGI like variables; and
  - a callback function that will be used by the application to send HTTP status code/message and HTTP headers to the server.
- and must return the response body to the server as strings wrapped in an iterable.

The application skeleton:

```
# This is our application object. It could have any name,
# except when using mod_wsgi where it must be "application"
def application( # It accepts two arguments:
    # environ points to a dictionary containing CGI like environment variables
    # which is filled by the server for each received request from the client
    environ,
    # start_response is a callback function supplied by the server
    # which will be used to send the HTTP status and headers to the server
    start_response):

  # build the response body possibly using the environ dictionary
  response_body = 'The request method was %s' % environ['REQUEST_METHOD']

  # HTTP response code and message
  status = '200 OK'

  # These are HTTP headers expected by the client.
  # They must be wrapped as a list of tupled pairs:
  # [(Header name, Header value)].
  response_headers = [('Content-Type', 'text/plain'),
                      ('Content-Length', str(len(response_body)))]

  # Send them to the server using the supplied function
  start_response(status, response_headers)

  # Return the response body.
  # Notice it is wrapped in a list although it could be any iterable.
  return [response_body]
```

This code will not run because it lacks the server instantiation step. Next page will show how to do it.

3. Environment Dictionary

The environment dictionary will contain CGI like variables and will be populated by the server at each request from the client. This script will output the whole dictionary:

```
#! /usr/bin/env python

# Our tutorial's WSGI server
from wsgiref.simple_server import make_server

def application(environ, start_response):

  # Sorting and stringifying the environment key, value pairs
  response_body = ['%s: %s' % (key, value)
                   for key, value in sorted(environ.items())]
  response_body = '\n'.join(response_body)

  status = '200 OK'
  response_headers = [('Content-Type', 'text/plain'),
                      ('Content-Length', str(len(response_body)))]
  start_response(status, response_headers)

  return [response_body]

# Instantiate the WSGI server.
# It will receive the request, pass it to the application
# and send the application's response to the client
httpd = make_server(
```

```
  'localhost', # The host name.
  8051, # A port number where to wait for the request.
  application # Our application object name, in this case a function.
  )

# Wait for a single request, serve it and quit.
httpd.handle_request()
```

To run this script save it as environment.py, open the terminal, navigate to the directory where it was saved and type python environment.py at the command line.

If in Windows it is necessary first to add the path to python.exe to the system environment variable Path. By the way if you can use Linux in instead of Windows then do it. It will save some pain.

Now go to the browser and type at the address bar:

```
http://localhost:8051/
```

4. Response Iterable

If the last script worked change the return line from:

```
return [response_body]
```

to:

```
 return response_body
```

Then run it again. Noticed it slower? **What happened is that the server iterated over the string sending a single byte at a time to the client. So don't forget to wrap the response in a better performance iterable like a list.** If the iterable yields more than one string the content_length will be the sum of all the string's lengths like in this script:

```
#! /usr/bin/env python

from wsgiref.simple_server import make_server

def application(environ, start_response):

  response_body = ['%s: %s' % (key, value)
              for key, value in sorted(environ.items())]
  response_body = '\n'.join(response_body)

  # Response_body has now more than one string
  response_body = ['The Beggining\n',
              '*' * 30 + '\n',
              response_body,
              '\n' + '*' * 30 ,
              '\nThe End']

  # So the content-lenght is the sum of all string's lengths
  content_length = 0
  for s in response_body:
    content_length += len(s)

  status = '200 OK'
  response_headers = [('Content-Type', 'text/plain'),
            ('Content-Length', str(content_length))]
  start_response(status, response_headers)

  return response_body

httpd = make_server('localhost', 8051, application)
httpd.handle_request()
```

5. Parsing the Request - Get

Run the above py file again and this time call it like this:

```
http://localhost:8051/?age=10&hobbies=software&hobbies=tunning
```

Notice the *QUERY_STRING* and the *REQUEST_METHOD* variables in the environ dictionary. When the request method is GET the form variables will be sent in the URL in the part called query string, that is, everything after the ?. The value of the query string here is *age=10&hobbies=software&hobbies=tunning*. Notice the hobbies variable appears two times. It can happen when there are checkboxes in the form or when the user type the same variable more than once in the URL.

It is possible to write code to parse the query string and retrieve those values but it is easier to use the CGI module's *parse_qs* function which returns a dictionary with the values as lists.

Always beware of the user input. Sanitise it to avoid script injection. The CGI's *escape* function can be used for that.

For the next script to work it must be saved as parsing_get.wsgi as that is the value of the action attribute of the form. The wsgi extension is the most used for the main script when using mod_wsgi.

```
#!/usr/bin/env python

from wsgiref.simple_server import make_server
from cgi import parse_qs, escape

html = """
<html>
<body>
  <form method="get" action="parsing_get.wsgi">
    <p>
      Age: <input type="text" name="age">
      </p>
    <p>
      Hobbies:
      <input name="hobbies" type="checkbox" value="software"> Software
      <input name="hobbies" type="checkbox" value="tunning"> Auto Tunning
      </p>
```

```
    <p>
      <input type="submit" value="Submit">
      </p>
    </form>
  <p>
    Age: %s<br>
    Hobbies: %s
    </p>
  </body>
</html>"""

def application(environ, start_response):

  # Returns a dictionary containing lists as values.
  d = parse_qs(environ['QUERY_STRING'])

  # In this idiom you must issue a list containing a default value.
  age = d.get('age', [''])[0] # Returns the first age value.
  hobbies = d.get('hobbies', []) # Returns a list of hobbies.

  # Always escape user input to avoid script injection
  age = escape(age)
  hobbies = [escape(hobby) for hobby in hobbies]

  response_body = html % (age or 'Empty',
            ', '.join(hobbies or ['No Hobbies']))

  status = '200 OK'

  # Now content type is text/html
  response_headers = [('Content-Type', 'text/html'),
              ('Content-Length', str(len(response_body)))]
  start_response(status, response_headers)

  return [response_body]

httpd = make_server('localhost', 8051, application)
# Now it is serve_forever() in instead of handle_request().
# In Windows you can kill it in the Task Manager (python.exe).
# In Linux a Ctrl-C will do it.
httpd.serve_forever()
```

6. Parsing the Request - Post

When the request method is POST the query string will be sent in the HTTP request body in instead of in the URL. The request body is in the WSGI server supplied wsgi.input file like environment variable.

It is necessary to know the response body size as an integer to read it from wsgi.input. WSGI specification says the CONTENT_LENGTH variable, which holds the body size, may be empty or missing so read it in a try/except block.

This script should be saved as parsing_post.wsgi

```
#!/usr/bin/env python

from wsgiref.simple_server import make_server
from cgi import parse_qs, escape

html = """
<html>
<body>
  <form method="post" action="parsing_post.wsgi">
    <p>
      Age: <input type="text" name="age">
      </p>
    <p>
      Hobbies:
      <input name="hobbies" type="checkbox" value="software"> Software
      <input name="hobbies" type="checkbox" value="tunning"> Auto Tunning
      </p>
    <p>
      <input type="submit" value="Submit">
      </p>
    </form>
  <p>
    Age: %s<br>
    Hobbies: %s
    </p>
  </body>
</html>
"""

def application(environ, start_response):

  # the environment variable CONTENT_LENGTH may be empty or missing
  try:
    request_body_size = int(environ.get('CONTENT_LENGTH', 0))
  except (ValueError):
    request_body_size = 0

  # When the method is POST the query string will be sent
```

```
    # in the HTTP request body which is passed by the WSGI server
    # in the file like wsgi.input environment variable.
    request_body = environ['wsgi.input'].read(request_body_size)
    d = parse_qs(request_body)

    age = d.get('age', [''])[0] # Returns the first age value.
    hobbies = d.get('hobbies', []) # Returns a list of hobbies.

    # Always escape user input to avoid script injection
    age = escape(age)
    hobbies = [escape(hobby) for hobby in hobbies]

    response_body = html % (age or 'Empty',
                ', '.join(hobbies or ['No Hobbies']))

    status = '200 OK'

    response_headers = [('Content-Type', 'text/html'),
                    ('Content-Length', str(len(response_body)))]
    start_response(status, response_headers)

    return [response_body]

httpd = make_server('localhost', 8051, application)
httpd.serve_forever()
```