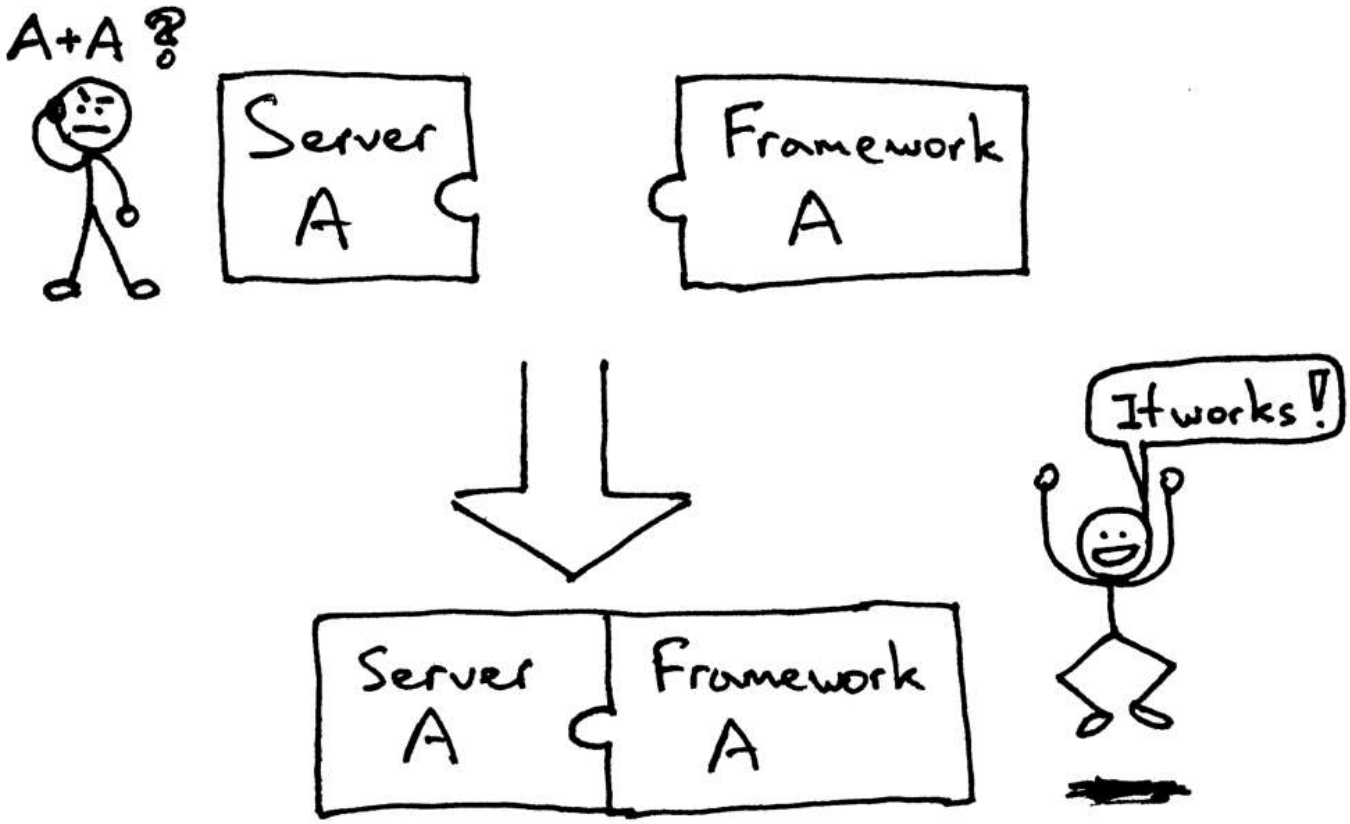
Let's Build A Web Server. Part 2. (https://ruslanspivak.com/lsbaws-part2/)

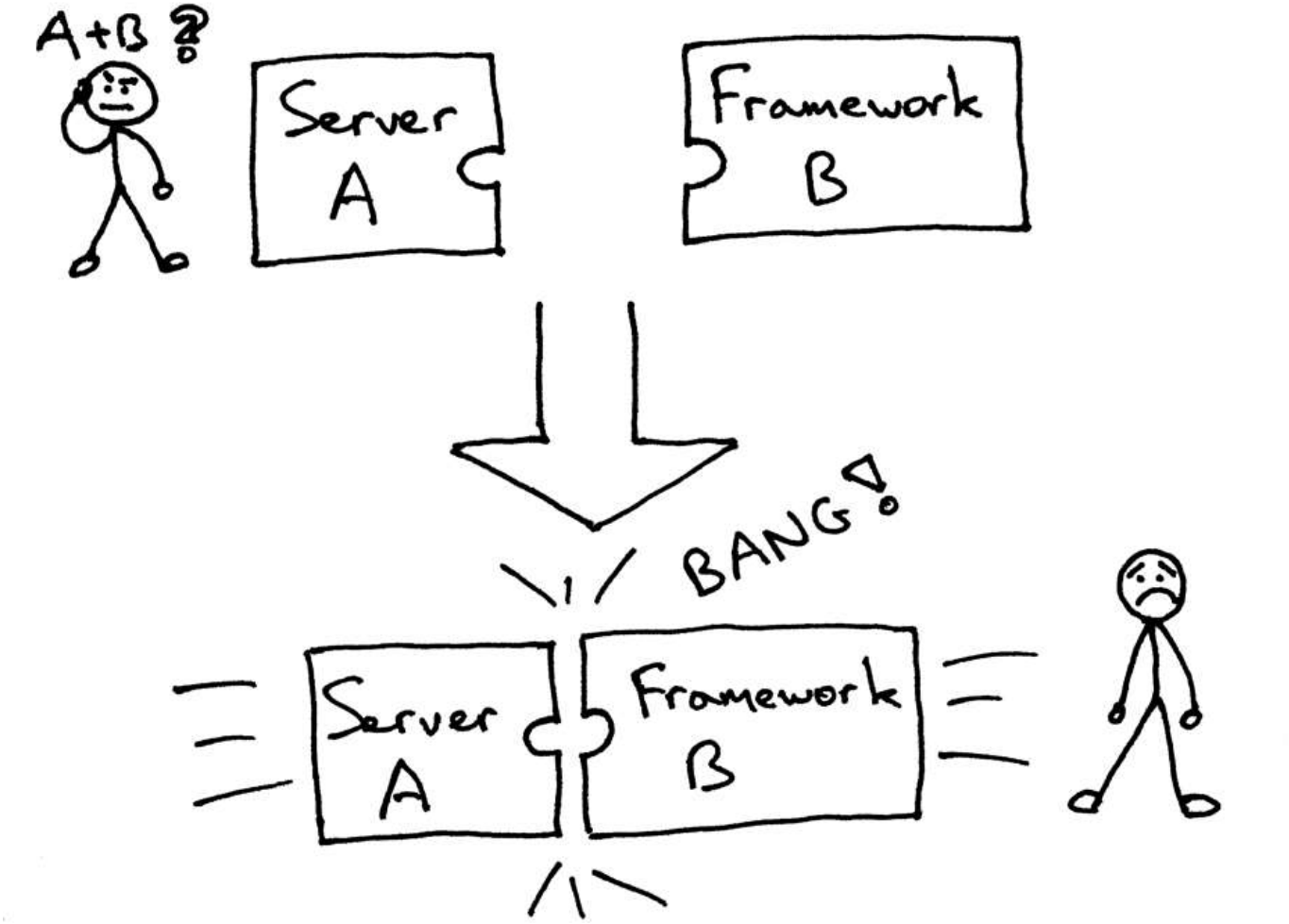
Date Mon, April 06, 2015

Remember, in Part 1 (http://ruslanspivak.com/lsbaws-part1/) I asked you a question: "How do you run a Django application, Flask application, and Pyramid application under your freshly minted Web server without making a single change to the server to accommodate all those different Web frameworks?" Read on to find out the answer.

In the past, your choice of a Python Web framework would limit your choice of usable Web servers, and vice versa. If the framework and the server were designed to work together, then you were okay:



But you could have been faced (and maybe you were) with the following problem when trying to combine a server and a framework that weren't designed to work together:

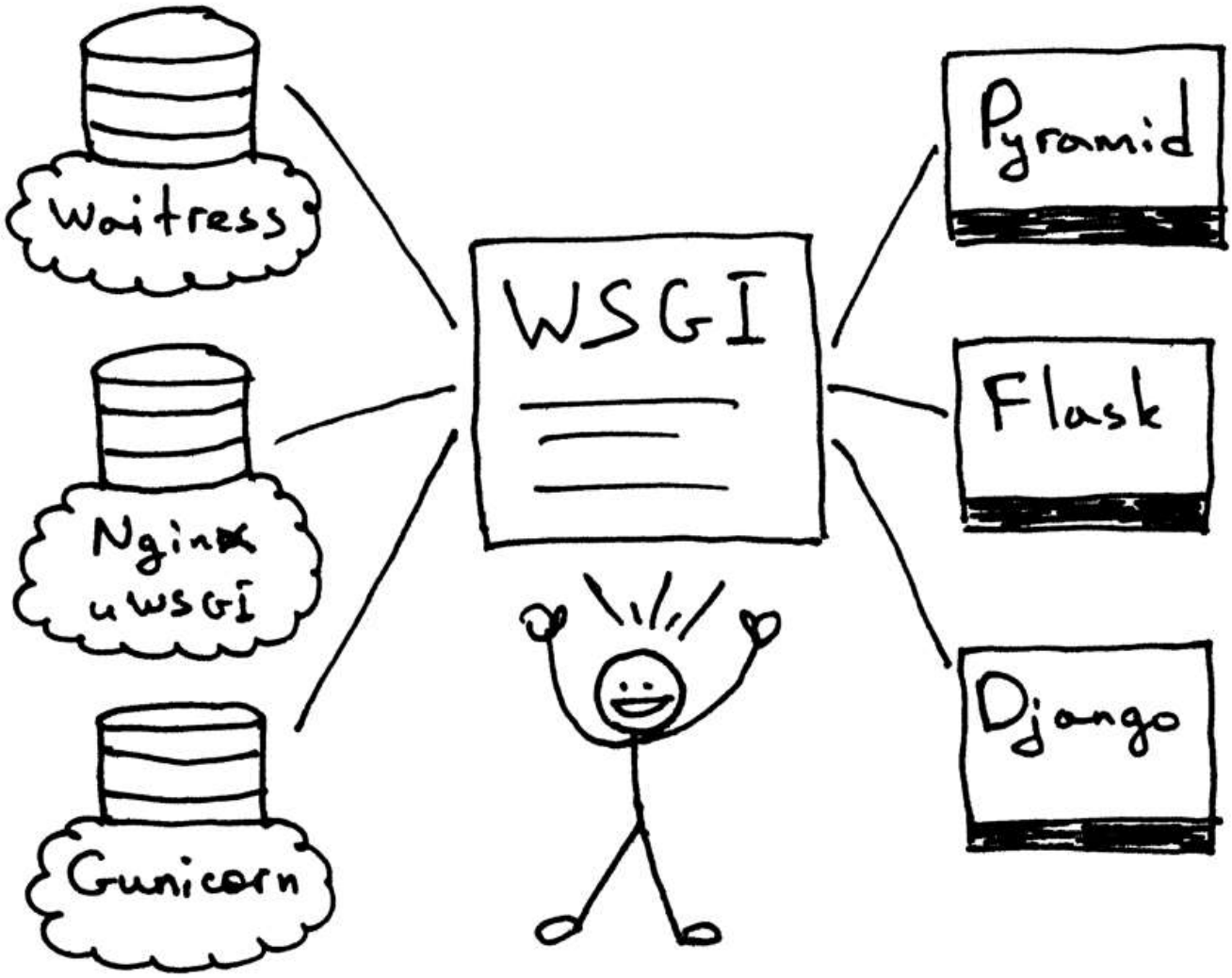


Basically you had to use what worked together and not what you might have wanted to use.

So, how do you then make sure that you can run your Web server with multiple Web frameworks without making code changes either to the Web server or to the Web frameworks? And the answer to that problem became the **Python Web Server Gateway Interface** (or WSGI (https://www.python.org/dev/peps/pep-0333/) for short, pronounced *"wizgy"*).

WSGI (https://www.python.org/dev/peps/pep-0333/) allowed developers to separate choice of a Web framework from choice of a Web server. Now you can actually mix and match Web servers and Web frameworks and choose a pairing that suits your needs. You can run Django (https://www.djangoproject.com/), Flask (http://flask.pocoo.org/), or Pyramid (http://trypyramid.com/), for example, with Gunicorn (http://gunicorn.org/) or Nginx/uWSGI (http://uwsgi-

docs.readthedocs.org) or Waitress (http://waitress.readthedocs.org). Real mix and match, thanks to the WSGI support in both servers and frameworks:



So, WSGI (https://www.python.org/dev/peps/pep-0333/) is the answer to the question I asked you in Part 1 (http://ruslanspivak.com/lsbaws-part1/) and repeated at the beginning of this article. Your Web server must implement the server portion of a WSGI interface and all modern Python Web Frameworks already implement the framework side of the WSGI interface, which allows you to use them with your Web server without ever modifying your server's code to accommodate a particular Web framework.

Now you know that WSGI support by Web servers and Web frameworks allows you to choose a pairing that suits you, but it is also beneficial to server and framework developers because they can focus on their preferred area of

specialization and not step on each other's toes. Other languages have similar interfaces too: Java, for example, has Servlet API (http://en.wikipedia.org/wiki/Java_servlet) and Ruby has Rack (http://en.wikipedia.org/wiki/Rack_%28web_server_interface%29).

It's all good, but I bet you are saying: "Show me the code!" Okay, take a look at this pretty minimalistic WSGI server implementation:

```
# Tested with Python 2.7.9, Linux & Mac OS X
import socket
import StringIO
import sys


class WSGIServer(object):

    address_family = socket.AF_INET
    socket_type = socket.SOCK_STREAM
    request_queue_size = 1

    def __init__(self, server_address):
        # Create a listening socket
        self.listen_socket = listen_socket = socket.socket(
            self.address_family,
            self.socket_type
        )
        # Allow to reuse the same address
        listen_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        # Bind
        listen_socket.bind(server_address)
        # Activate
        listen_socket.listen(self.request_queue_size)
        # Get server host name and port
        host, port = self.listen_socket.getsockname()[:2]
        self.server_name = socket.getfqdn(host)
        self.server_port = port
        # Return headers set by Web framework/Web application
        self.headers_set = []

    def set_app(self, application):
        self.application = application

    def serve_forever(self):
        listen_socket = self.listen_socket
        while True:
            # New client connection
            self.client_connection, client_address = listen_socket.accept()
            # Handle one request and close the client connection. Then
```

```
            # loop over to wait for another client connection
            self.handle_one_request()

    def handle_one_request(self):
        self.request_data = request_data = self.client_connection.recv(1024
        # Print formatted request data a la 'curl -v'
        print(''.join(
            '< {line}\n'.format(line=line)
            for line in request_data.splitlines()
        ))

        self.parse_request(request_data)

        # Construct environment dictionary using request data
        env = self.get_environ()

        # It's time to call our application callable and get
        # back a result that will become HTTP response body
        result = self.application(env, self.start_response)

        # Construct a response and send it back to the client
        self.finish_response(result)

    def parse_request(self, text):
        request_line = text.splitlines()[0]
        request_line = request_line.rstrip('\r\n')
        # Break down the request line into components
        (self.request_method,  # GET
         self.path,            # /hello
         self.request_version  # HTTP/1.1
         ) = request_line.split()

    def get_environ(self):
        env = {}
        # The following code snippet does not follow PEP8 conventions
        # but it's formatted the way it is for demonstration purposes
        # to emphasize the required variables and their values
        #
        # Required WSGI variables
        env['wsgi.version']      = (1, 0)
```

```
        env['wsgi.url_scheme']   = 'http'
        env['wsgi.input']        = StringIO.StringIO(self.request_data)
        env['wsgi.errors']       = sys.stderr
        env['wsgi.multithread']  = False
        env['wsgi.multiprocess'] = False
        env['wsgi.run_once']     = False
        # Required CGI variables
        env['REQUEST_METHOD']    = self.request_method    # GET
        env['PATH_INFO']         = self.path              # /hello
        env['SERVER_NAME']       = self.server_name       # localhost
        env['SERVER_PORT']       = str(self.server_port)  # 8888
        return env

    def start_response(self, status, response_headers, exc_info=None):
        # Add necessary server headers
        server_headers = [
            ('Date', 'Tue, 31 Mar 2015 12:54:48 GMT'),
            ('Server', 'WSGIServer 0.2'),
        ]
        self.headers_set = [status, response_headers + server_headers]
        # To adhere to WSGI specification the start_response must return
        # a 'write' callable. We simplicity's sake we'll ignore that detail
        # for now.
        # return self.finish_response

    def finish_response(self, result):
        try:
            status, response_headers = self.headers_set
            response = 'HTTP/1.1 {status}\r\n'.format(status=status)
            for header in response_headers:
                response += '{0}: {1}\r\n'.format(*header)
            response += '\r\n'
            for data in result:
                response += data
            # Print formatted response data a la 'curl -v'
            print(''.join(
                '> {line}\n'.format(line=line)
                for line in response.splitlines()
            ))
            self.client_connection.sendall(response)
```

```
        finally:
            self.client_connection.close()


SERVER_ADDRESS = (HOST, PORT) = '', 8888


def make_server(server_address, application):
    server = WSGIServer(server_address)
    server.set_app(application)
    return server


if __name__ == '__main__':
    if len(sys.argv) < 2:
        sys.exit('Provide a WSGI application object as module:callable')
    app_path = sys.argv[1]
    module, application = app_path.split(':')
    module = __import__(module)
    application = getattr(module, application)
    httpd = make_server(SERVER_ADDRESS, application)
    print('WSGIServer: Serving HTTP on port {port} ...\n'.format(port=PORT))
    httpd.serve_forever()
```

It's definitely bigger than the server code in Part 1 (http://ruslanspivak.com/lsbaws-part1/), but it's also small enough (just under 150 lines) for you to understand without getting bogged down in details. The above server also does more - it can run your basic Web application written with your beloved Web framework, be it Pyramid, Flask, Django, or some other Python WSGI framework.

Don't believe me? Try it and see for yourself. Save the above code as *webserver2.py* or download it directly from GitHub (https://github.com/rspivak/lsbaws/blob/master/part2/webserver2.py). If you try to run it without any parameters it's going to complain and exit.

```
$ python webserver2.py
Provide a WSGI application object as module:callable
```

It really wants to serve your Web application and that's where the fun begins. To run the server the only thing you need installed is Python. But to run applications written with Pyramid, Flask, and Django you need to install those frameworks first. Let's install all three of them. My preferred method is by using virtualenv (https://virtualenv.pypa.io). Just follow the steps below to create and activate a virtual environment and then install all three Web frameworks.

```
$ [sudo] pip install virtualenv
$ mkdir ~/envs
$ virtualenv ~/envs/lsbaws/
$ cd ~/envs/lsbaws/
$ ls
bin  include  lib
$ source bin/activate
(lsbaws) $ pip install pyramid
(lsbaws) $ pip install flask
(lsbaws) $ pip install django
```

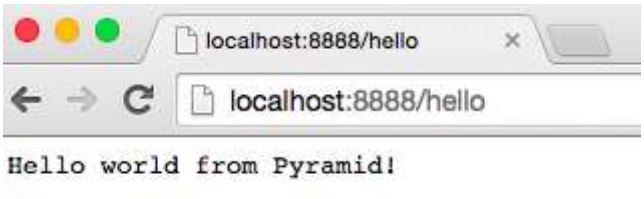
At this point you need to create a Web application. Let's start with Pyramid (http://trypyramid.com/) first. Save the following code as *pyramidapp.py* to the same directory where you saved *webserver2.py* or download the file directly from GitHub (https://github.com/rspivak/lsbaws/blob/master/part2/pyramidapp.py):

```
from pyramid.config import Configurator
from pyramid.response import Response


def hello_world(request):
    return Response(
        'Hello world from Pyramid!\n',
        content_type='text/plain',
    )

config = Configurator()
config.add_route('hello', '/hello')
config.add_view(hello_world, route_name='hello')
app = config.make_wsgi_app()
```

Now you're ready to serve your Pyramid application with your very own Web server:

```
(lsbaws) $ python webserver2.py pyramidapp:app
WSGIServer: Serving HTTP on port 8888 ...
```

You just told your server to load the *'app'* callable from the python module *'pyramidapp'* Your server is now ready to take requests and forward them to your Pyramid application. The application only handles one route now: the */hello* route. Type http://localhost:8888/hello (http://localhost:8888/hello) address into your browser, press Enter, and observe the result:



You can also test the server on the command line using the *'curl'* utility:

```
$ curl -v http://localhost:8888/hello (http://localhost:8888/hello)
...
```

Check what the server and *curl* prints to standard output.

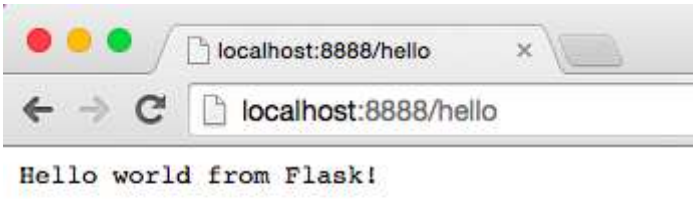
Now onto Flask (http://flask.pocoo.org/). Let's follow the same steps.

```
from flask import Flask
from flask import Response
flask_app = Flask('flaskapp')


@flask_app.route('/hello')
def hello_world():
    return Response(
        'Hello world from Flask!\n',
        mimetype='text/plain'
    )

app = flask_app.wsgi_app
```

Save the above code as *flaskapp.py* or download it from GitHub (https://github.com/rspivak/lsbaws/blob/master/part2/flaskapp.py) and run the server as:

```
(lsbaws) $ python webserver2.py flaskapp:app
WSGIServer: Serving HTTP on port 8888 ...
```

Now type in the http://localhost:8888/hello (http://localhost:8888/hello) into your browser and press Enter:



Again, try *'curl'* and see for yourself that the server returns a message generated by the Flask application:

```
$ curl -v http://localhost:8888/hello (http://localhost:8888/hello)
...
```

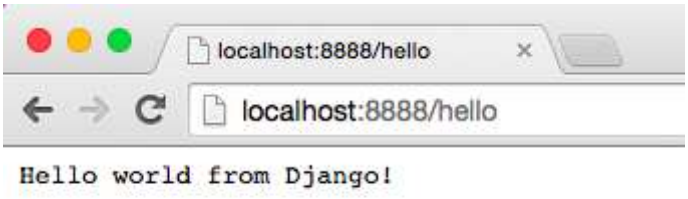
Can the server also handle a Django (https://www.djangoproject.com/) application? Try it out! It's a little bit more involved, though, and I would recommend cloning the whole repo and use djangoapp.py (https://github.com/rspivak/lsbaws/blob/master/part2/djangoapp.py), which is part of the GitHub repository (https://github.com/rspivak/lsbaws/). Here is the source code which basically adds the Django *'helloworld'* project (pre-created using Django's *django-admin.py startproject* command) to the current Python path and then imports the project's WSGI application.

```
import sys
sys.path.insert(0, './helloworld')
from helloworld import wsgi


app = wsgi.application
```

Save the above code as *djangoapp.py* and run the Django application with your Web server:

```
(lsbaws) $ python webserver2.py djangoapp:app
WSGIServer: Serving HTTP on port 8888 ...
```

Type in the following address and press Enter:



And as you've already done a couple of times before, you can test it on the command line, too, and confirm that it's the Django application that handles your requests this time around:

```
$ curl -v http://localhost:8888/hello (http://localhost:8888/hello)
...
```

Did you try it? Did you make sure the server works with those three frameworks? If not, then please do so. Reading is important, but this series is about rebuilding and that means you need to get your hands dirty. Go and try it. I will wait for you, don't worry. No seriously, you must try it and, better yet, retype everything yourself and make sure that it works as expected.

Okay, you've experienced the power of WSGI: it allows you to mix and match your Web servers and Web frameworks. WSGI provides a minimal interface between Python Web servers and Python Web Frameworks. It's very simple and it's easy to implement on both the server and the framework side. The following code snippet shows the server and the framework side of the interface:

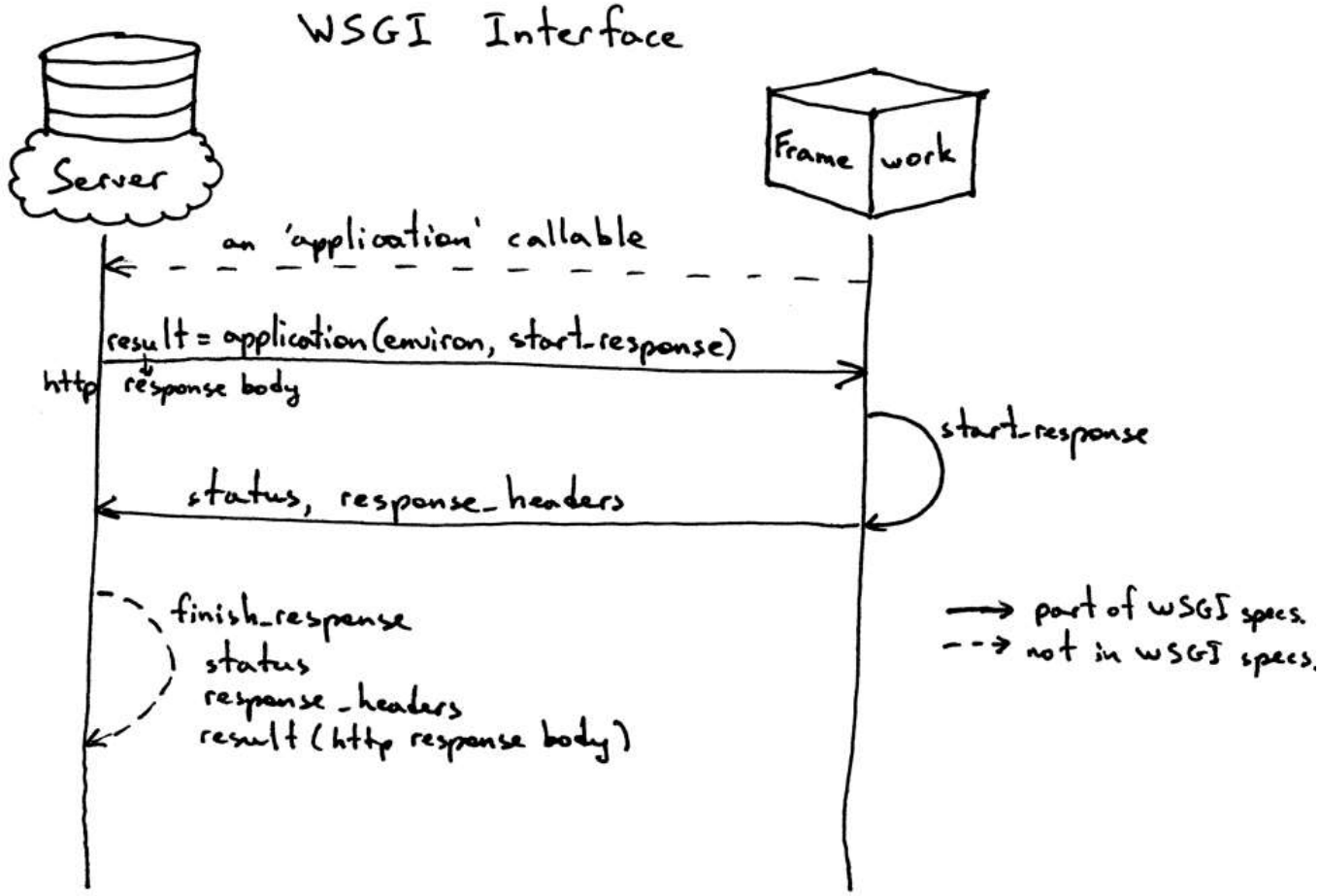
```
def run_application(application):
    """Server code."""
    # This is where an application/framework stores
    # an HTTP status and HTTP response headers for the server
    # to transmit to the client
    headers_set = []
    # Environment dictionary with WSGI/CGI variables
    environ = {}

    def start_response(status, response_headers, exc_info=None):
        headers_set[:] = [status, response_headers]

    # Server invokes the ‘application' callable and gets back the
    # response body
    result = application(environ, start_response)
    # Server builds an HTTP response and transmits it to the client
    …

def app(environ, start_response):
    """A barebones WSGI app."""
    start_response('200 OK', [('Content-Type', 'text/plain')])
    return ['Hello world!']

run_application(app)
```

Here is how it works:

1. The framework provides an *‘application’* callable (The WSGI specification doesn't prescribe how that should be implemented)
2. The server invokes the *‘application’* callable for each request it receives from an HTTP client. It passes a dictionary *‘environ’* containing WSGI/CGI variables and a *‘start_response’* callable as arguments to the *‘application’* callable.
3. The framework/application generates an HTTP status and HTTP response headers and passes them to the *‘start_response’* callable for the server to store them. The framework/application also returns a response body.
4. The server combines the status, the response headers, and the response body into an HTTP response and transmits it to the client (This step is not

part of the specification but it's the next logical step in the flow and I added it for clarity)

And here is a visual representation of the interface:



So far, you've seen the Pyramid, Flask, and Django Web applications and you've seen the server code that implements the server side of the WSGI specification. You've even seen the barebones WSGI application code snippet that doesn't use any framework.

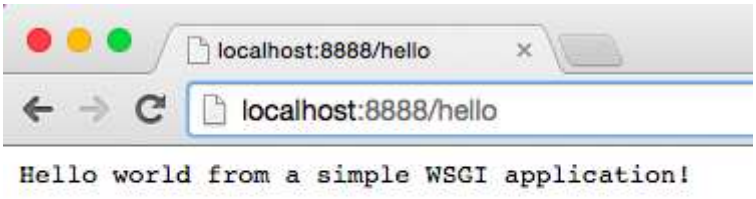
The thing is that when you write a Web application using one of those frameworks you work at a higher level and don't work with WSGI directly, but I know you're curious about the framework side of the WSGI interface, too because you're reading this article. So, let's create a minimalistic WSGI Web application/Web framework without using Pyramid, Flask, or Django and run it with your server:

```
def app(environ, start_response):
    """A barebones WSGI application.

    This is a starting point for your own Web framework :)
    """
    status = '200 OK'
    response_headers = [('Content-Type', 'text/plain')]
    start_response(status, response_headers)
    return ['Hello world from a simple WSGI application!\n']
```

Again, save the above code in *wsgiapp.py* file or download it from GitHub (https://github.com/rspivak/lsbaws/blob/master/part2/wsgiapp.py) directly and run the application under your Web server as:

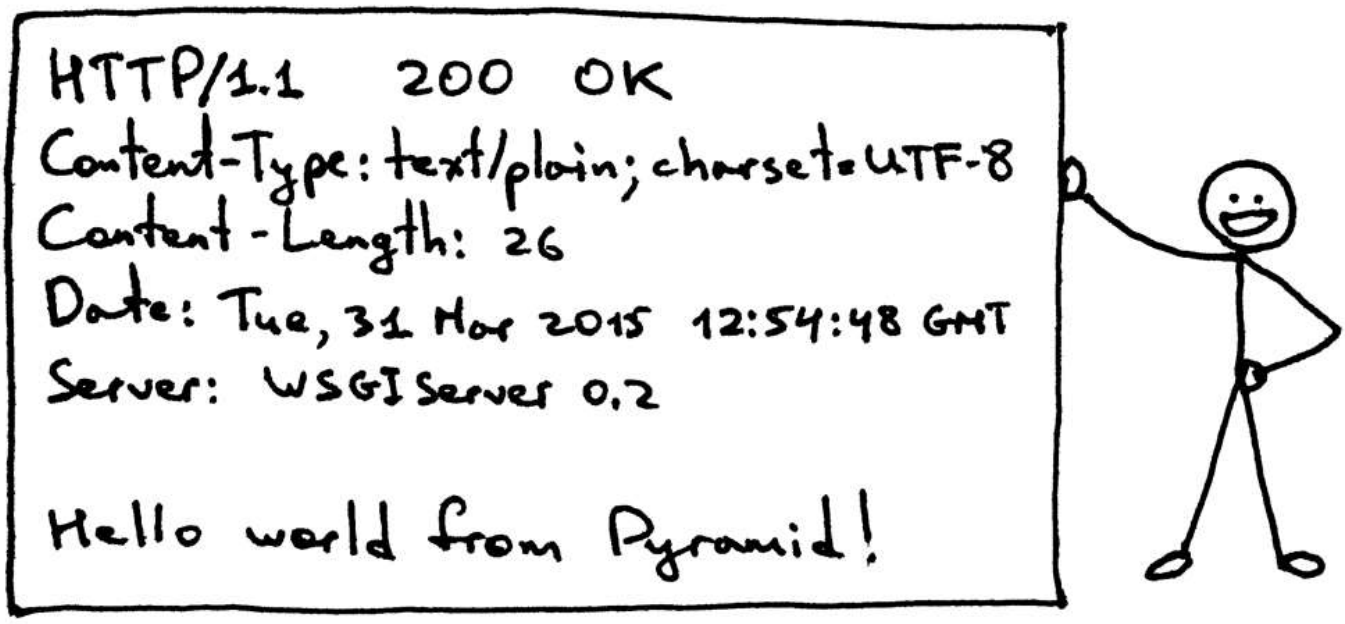
```
(lsbaws) $ python webserver2.py wsgiapp:app
WSGIServer: Serving HTTP on port 8888 ...
```

Type in the following address and press Enter. This is the result you should see:



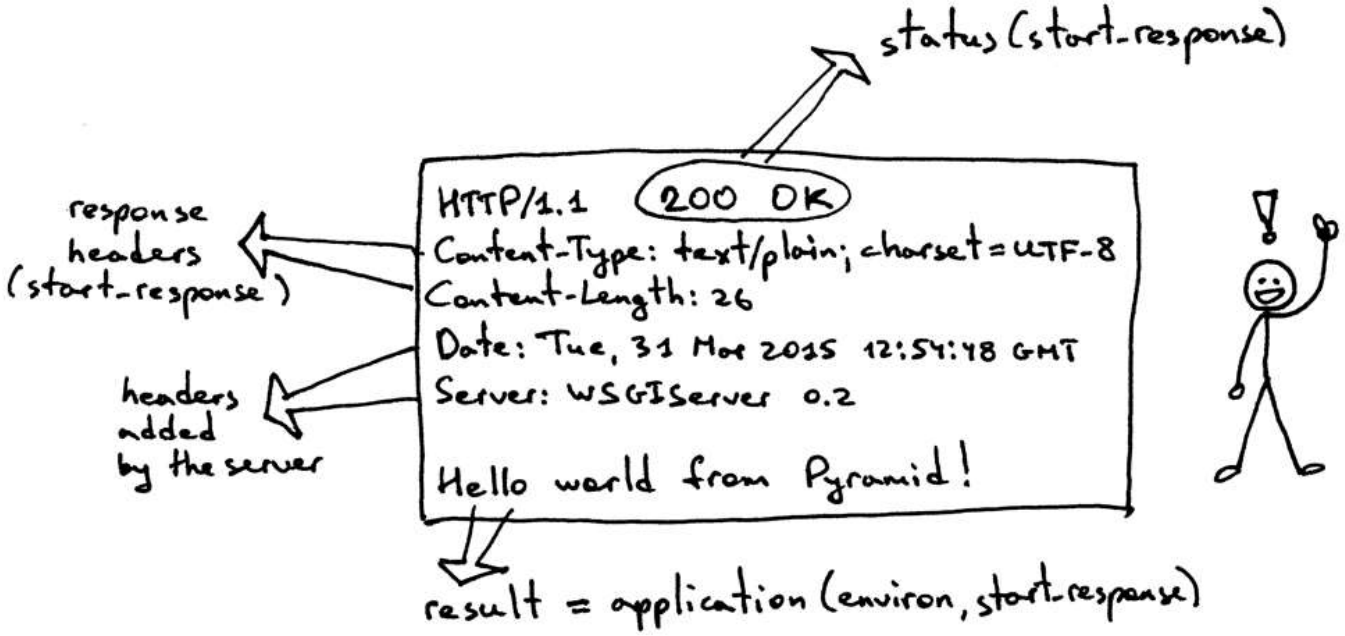
You just wrote your very own minimalistic WSGI Web framework while learning about how to create a Web server! Outrageous.

Now, let's get back to what the server transmits to the client. Here is the HTTP response the server generates when you call your Pyramid application using an HTTP client:

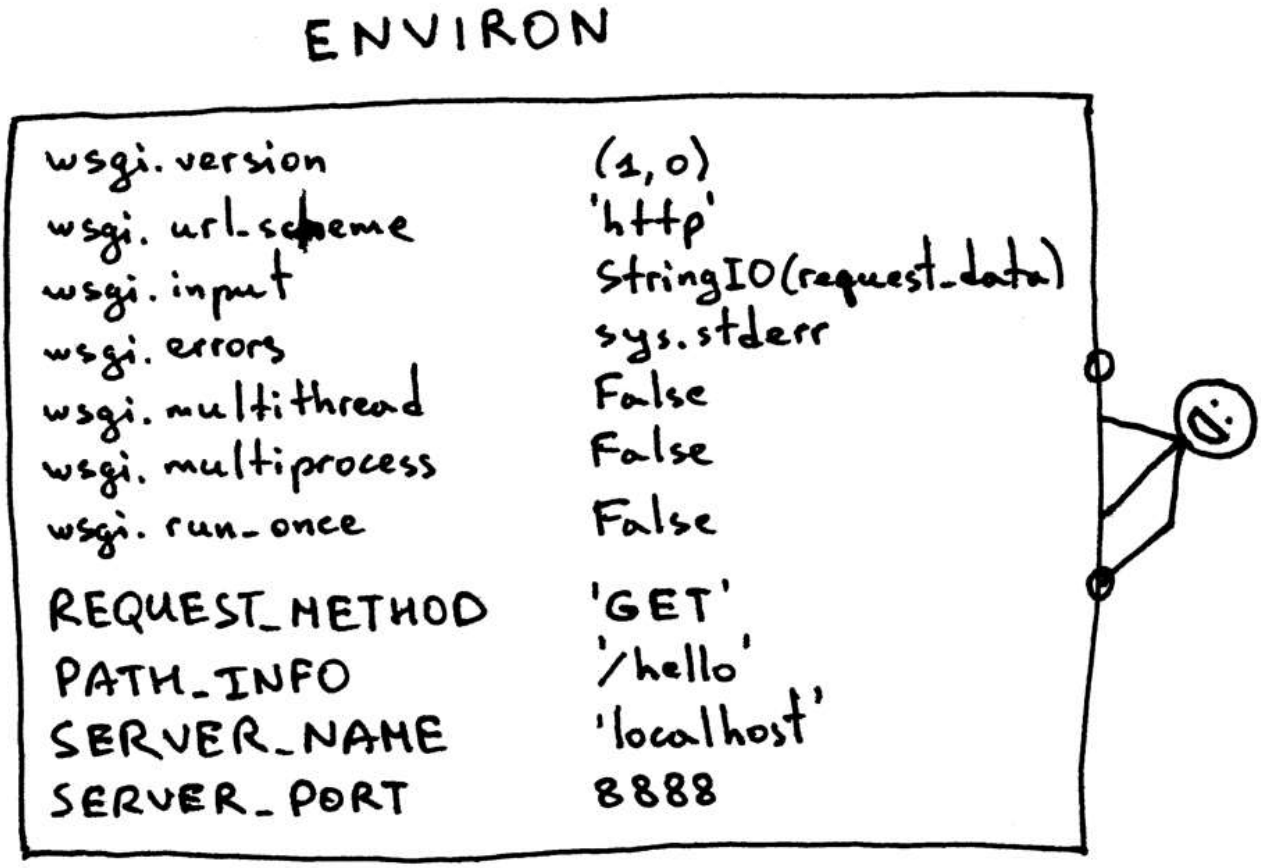


The response has some familiar parts that you saw in Part 1 (http://ruslanspivak.com/lsbaws-part1/) but it also has something new. It has, for example, four HTTP headers (http://en.wikipedia.org/wiki/List_of_HTTP_header_fields) that you haven't seen before: *Content-Type*, *Content-Length*, *Date*, and *Server*. Those are the headers that a response from a Web server generally should have. None of them are strictly required, though. The purpose of the headers is to transmit additional information about the HTTP request/response.

Now that you know more about the WSGI interface, here is the same HTTP response with some more information about what parts produced it:



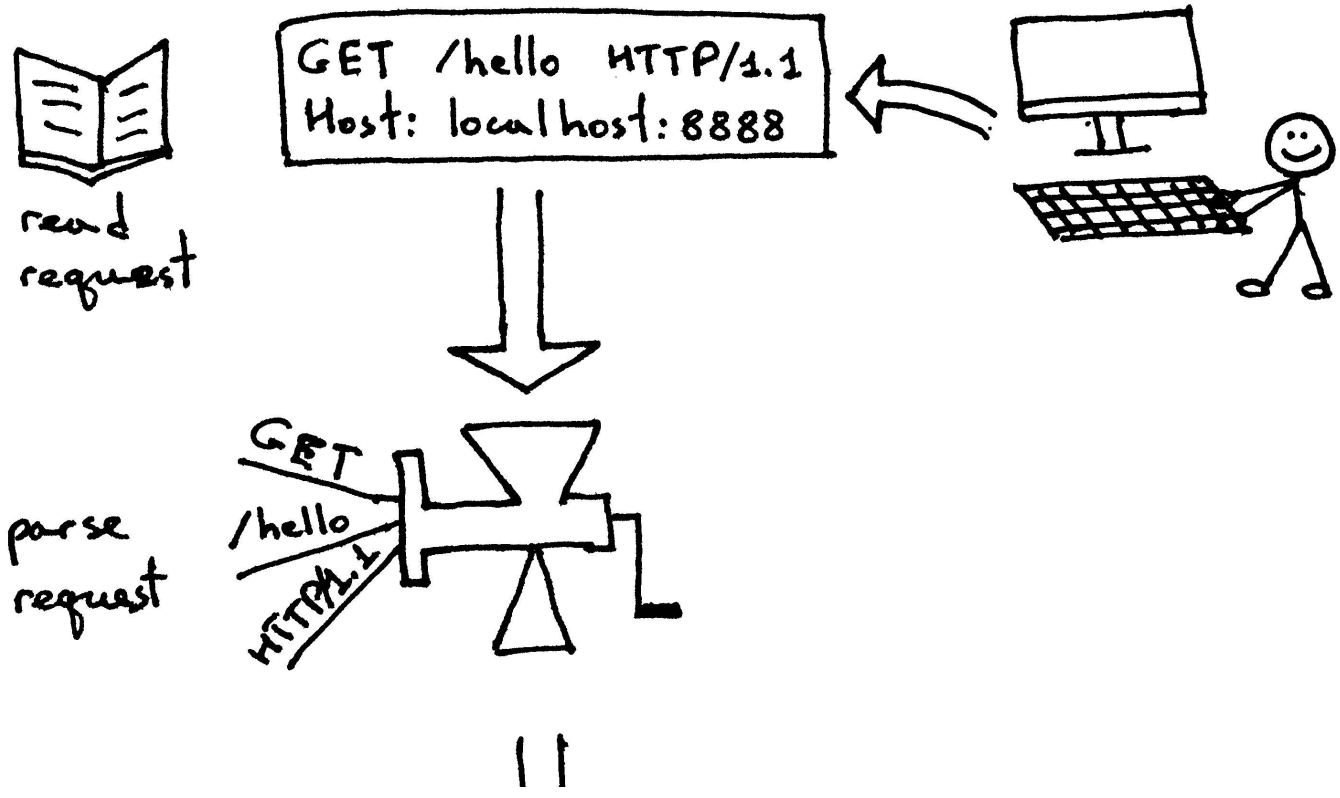
I haven't said anything about the **'environ'** dictionary yet, but basically it's a Python dictionary that must contain certain WSGI and CGI variables prescribed by the WSGI specification. The server takes the values for the dictionary from the HTTP request after parsing the request. This is what the contents of the dictionary look like:

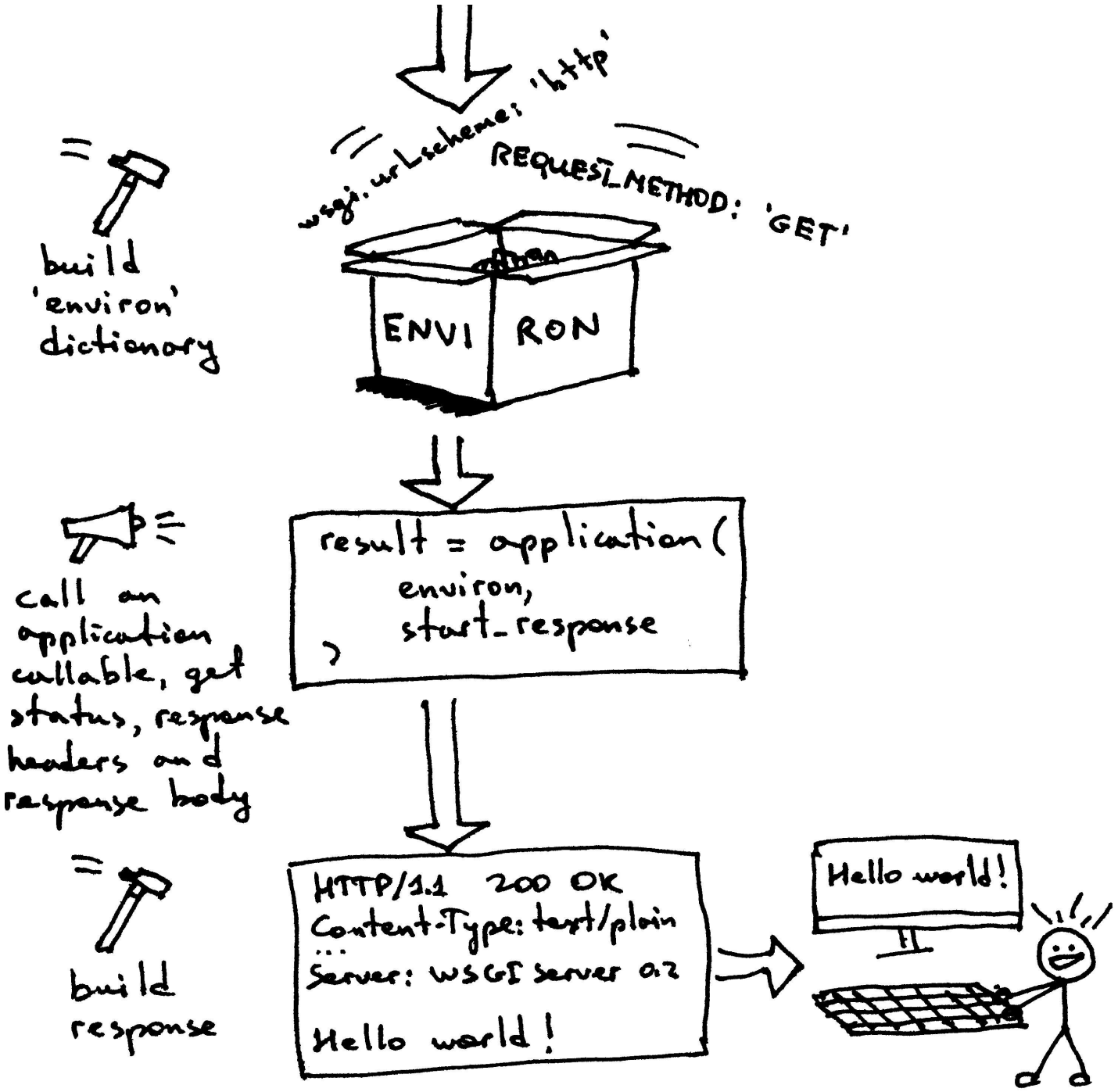


A Web framework uses the information from that dictionary to decide which view to use based on the specified route, request method etc., where to read the request body from and where to write errors, if any.

By now you've created your own WSGI Web server and you've made Web applications written with different Web frameworks. And, you've also created your barebones Web application/Web framework along the way. It's been a heck of a journey. Let's recap what your WSGI Web server has to do to serve requests aimed at a WSGI application:

- First, the server starts and loads an *'application'* callable provided by your Web framework/application
- Then, the server reads a request
- Then, the server parses it
- Then, it builds an *'environ'* dictionary using the request data
- Then, it calls the *'application'* callable with the *'environ'* dictionary and a *'start_response'* callable as parameters and gets back a response body.
- Then, the server constructs an HTTP response using the data returned by the call to the *'application'* object and the status and response headers set by the *'start_response'* callable.
- And finally, the server transmits the HTTP response back to the client





That's about all there is to it. You now have a working WSGI server that can serve basic Web applications written with WSGI compliant Web frameworks like Django (https://www.djangoproject.com/), Flask (http://flask.pocoo.org/), Pyramid (http://trypyramid.com/), or your very own WSGI framework. The best part is that the server can be used with multiple Web frameworks without any changes to the server code base. Not bad at all.

Before you go, here is another question for you to think about, *"How do you make your server handle more than one request at a time?"*

Stay tuned and I will show you a way to do that in Part 3. Cheers!

BTW, I'm writing a book "Let's Build A Web Server: First Steps" that explains how to write a basic web server from scratch and goes into more detail on topics I just covered. Subscribe to the mailing list to get the latest updates about the book and the release date.

Enter Your First Name *

Enter Your Best Email *

Get Updates!

All articles in this series:

- Let's Build A Web Server. Part 1. (http://ruslanspivak.com/lsbaws-part1/)
- Let's Build A Web Server. Part 2. (http://ruslanspivak.com/lsbaws-part2/)
- Let's Build A Web Server. Part 3. (http://ruslanspivak.com/lsbaws-part3/)

Comments

57 Comments **Ruslan's Blog** **Rahul Vats**

Recommend 19 Tweet Share **Sort by Best**

Join the discussion…

Erik • a year ago

I have the same problem:
Traceback (most recent call last):
File "webserver2.py", line 143, in <module>
httpd.serve_forever()
File "webserver2.py", line 42, in serve_forever
self.handle_one_request()
File "webserver2.py", line 52, in handle_one_request

self.parse_request(request_data)
File "webserver2.py", line 66, in parse_request
request_line = request_line.rstrip('\r\n')
TypeError: a bytes-like object is required, not 'str'

Please, how can i resolve it?

1 • Reply • Share ›

Fatih Kılıç ➚ Erik • a year ago

put b to string example
http_response = b"""\
HTTP/1.1 200 OK

Hello World!
"""

also you can do with

http_response = b"""\
HTTP/1.1 200 OK

%s
"""
http_response.encode('utf-8')

1 • Reply • Share ›

Gabriela de Lima • 2 months ago

Can anyone help me?
I have this error:

app_path = sys.argv[1]
--> 103 module, application = app_path.split(':')
104 module = __import__ (module)
105 application = getattr(module, application)

ValueError: not enough values to unpack (expected 2, got 1)

I tried to looking for some answers on Google but I can't find. Please!!

• Reply • Share ›

Gabriela de Lima ➚ Gabriela de Lima • a month ago

Solved

• Reply • Share ›

Bibek Chaudhary • 2 months ago

Hi, I am having some problem in POST request of flask. I am not able to send any headers and data in my flask app, can anyone help??

• Reply • Share ›

Aiden Gardner • 5 months ago

Hey, i was having a problem with importing StringIO in python 3.6: It says the module doesn't exist. Please help soon

C:\Users\Aiden\Desktop>webserver2.py pyramidapp:app
Traceback (most recent call last):
File "C:\Users\Aiden\Desktop\webserver2.py", line 3, in <module>
import StringIO
ModuleNotFoundError: No module named 'StringIO'

Reply • Share ›

Harun Tuncay ➚ Aiden Gardner • 5 months ago

try "from io import StringIO"

Reply • Share ›

James Uriu • 9 months ago

I am getting an import error when trying to run the django app. It says that there is no module named helloworld:

python webserver2.py djangoapp:app
Traceback (most recent call last):
File "webserver2.py", line 139, in <module>
module = __import__(module)
File "/home/jamesuriu/envs/lsbaws/djangoapp.py", line 3, in <module>
from helloworld import wsgi
ImportError: No module named helloworld

can someone help me resolve this?

Reply • Share ›

Tim Buckland ➚ James Uriu • 7 months ago

You need to create the Django helloworld app or clone it from the repository given in the tutorial.

If like me you are using Python 3, you can copy a Python 3 version of the helloworld app from here: https://github.com/Qarj/scr...

Just copy the whole folder 'helloworld'. Note that the url to invoke this example will be: http://localhost:8888/helloworld

Reply • Share ›

JimD ➚ James Uriu • 7 months ago

my last reply was based on where your pathing is from. with Django, I did clone the directory and made sure the paths were correct for access from my webserver and the app file. make sure to clone the directory with the django example.

Reply • Share ›

JimD ➚ James Uriu • 7 months ago

so 2 things, 1, save the app file in the same directory as the webserver2.py and 2 I have to be specific as to the shebang in the webserver2.py so that python was launching from the virtual environment. so the shebang should be an absolute path and look like this. #!/<your root="" dir="" path="">/envs/lsbaws/python. you can test the availability virtualenv setup and the config files by doing the following "envs/lsbaws/bin/python" to launch a python interpreter and then run django.config and see if you get any errors. If you do not then the above shebang entry should work for you. hope this helps. worked well for me.

^ v • Reply • Share ›

mujina • a year ago

If you need, here is a porting to python 3, with a lot of additional comments that explain the code line by line:
https://github.com/mujina93...

^ v • Reply • Share ›

Mira • a year ago

Please help! I try any possible solution in Internet but non was working or another error occurs.
"cannot import name patterns"
Request Method: GET
Request URL: http://SGOSPODI-DT.lcs.intern:8888/hello
Django Version: 1.11
Exception Type: ImportError

^ v • Reply • Share ›

Martin • 2 years ago

Well part 2 escalated Quickly

^ v • Reply • Share ›

Chao Li • 2 years ago

Thanks

^ v • Reply • Share ›

Vu Nguyen • 2 years ago

Can anybody help me solve the error: "WSGIServer object has no attribute 'request_method'

^ v • Reply • Share ›

y agabey • 2 years ago

Could anybody please port these codes to Python 3 . Thanks..

^ v • Reply • Share ›

Russell ➚ y agabey • 2 years ago

Check this
https://github.com/russell3...

1 ^ v • Reply • Share ›

Gabriela de Lima ➚ Russell • a month ago

Do you still have this? The link is out. It isn't work for me I will appreciate a lot if you help me with this.

• Reply • Share ›

Gabriela de Lima ➚ Russell • 2 months ago

Link is out of area. Do you still have it?

• Reply • Share ›

Stone sky • 2 years ago

when I type your code in my PyCharm, It says that importlib.import_module() is better than pure __import__().

• Reply • Share ›

Joseph • 3 years ago

Can someone please help me out with this. I am getting the following error when I try to run this from part 2.

File "//anaconda/lib/python2.7/site-packages/django/core/wsgi.py", line 13, in get_wsgi_application
django.setup()
File "//anaconda/lib/python2.7/site-packages/django/__init__.py", line 18, in setup
apps.populate(settings.INSTALLED_APPS)
File "//anaconda/lib/python2.7/site-packages/django/apps/registry.py", line 85, in populate
app_config = AppConfig.create(entry)
File "//anaconda/lib/python2.7/site-packages/django/apps/config.py", line 116, in create
mod = import_module(mod_path)
File "//anaconda/lib/python2.7/importlib/__init__.py", line 37, in import_module
__import__(name)
ImportError: No module named authdjango.contrib

Thanks!

• Reply • Share ›

whatbeg • 3 years ago

Thanks, but I encountered a problem, in the webserver2.py,

def parse_request(self, text):
request_line = text.splitlines()[0]
request_line = request_line.rstrip('\r\n')
.....

Question:

Traceback (most recent call last):
File "webserver.py", line 149, in <module>

```
httpd.serve_forever()
File "webserver.py", line 43, in serve_forever
self.handle_one_request()
File "webserver.py", line 53, in handle_one_request
self.parse_request(request_data)
File "webserver.py", line 71, in parse_request
request_line = request_line.rstrip('\r\n')
TypeError: a bytes-like object is required, not 'str'
```

and I comfirmed that 'text' is bytes, so how can apply following code to get the request line?

```
request_line = text.splitlines()[0]
request_line = request_line.rstrip('\r\n')
```

Reply • Share ›

efi • 3 years ago

I've already changed the following lines:

> env['wsgi.input'] = StringIO.StringIO(self.request_data)

to

< env['wsgi.input'] = StringIO(self.request_data)

> self.request_data = request_data = self.client_connection.recv(1024)

to

< self.request_data = request_data = self.client_connection.recv(1024).decode('utf-8')

< #self.client_connection.sendall(response)

to

see more

Reply • Share ›

efi • 3 years ago

what should I change in order it will work with Python3 as well?

Reply • Share ›

Manuel ➚ efi • 2 years ago

response += data.decode('utf-8')

Reply • Share ›

Sébastien • 3 years ago

Thanks a thousand times for your enlightments! I've just started getting interested in all these aspects to build my own WSGI app' and you made my life amazingly easier!!!
If I may just make a little remark, it seems to me that your usage of environ[wsgi.input] is not the same as in the spec. You put there the whole request whereas it's supposed to host the body of the request (which is empty in GET requests). As far as I understood, it is used when accessing POST requests in conjunction with environ[CONTENT_LENGTH] in the following way: envrion[wsgi.input].read(environ[CONTENT_LENGTH]).
Thanks again for your wonderful tutorial!

Reply • Share ›

pyhome • 3 years ago

真他喵的good，I have learnt a lot

Reply • Share ›

wei tu • 3 years ago

Thanks a lot. Article is great! How about adding "if(request_data==''') return" before calling self.parse_request(request_data) in webserver2.py. This may avoid an "index out of range" exception when run request_line = text.splitlines()[0]

Reply • Share ›

Sébastien ➚ wei tu • 3 years ago

Completely agree with this. I don't know exactly why but I keep receveing empty request that trigger the exception. Your solution solves the pb... but I'd still like to understand where these empty request come from.

Reply • Share ›

kroos • 3 years ago

真的太棒了，really great!

Reply • Share ›

Stuart Demarkles • 3 years ago

Did anyone else have the problem while trying to use the command "pip install pyramid"? Even after updating the pi I get the error message "Could not find a version that satisfies the requirement pyramid (from versions:)
No matching distribution found for pyramid"

Could anyone help?

Reply • Share ›

Noam Elfanbaum • 3 years ago

Great article!

Reply • Share ›

Yeshen Shang • 4 years ago

Has anybody encountered this problem? i get this when i run the server and make a request with

flask...

File "webserver.py", line 58, in handle_one_request

result = self.application(env, self.start_response)

TypeError: 'module' object is not callable

Thanks.

Reply • Share ›

victor • 4 years ago

Amazing article! Thank you for taking the time to do this.

Question... Why did you create 2 copies of the listening socket?
self.listen_socket = listen_socket = socket.socket(...)

Reply • Share ›

Eric So victor • 4 years ago

I'm not completely sure but I think its so that he can refer to the socket variable without prefixing the keyword self on to it. For example:

self.variable = variable = create_value()

Now self.variable and variable both refer to the same object and you can use the shorthand variable instead of self.variable within the function scope.

Reply • Share ›

mozillazg • 4 years ago

Great article! And the Chinese translation is here: http://mozillazg.com/2015/0...

Reply • Share ›

rspivak Mod mozillazg • 4 years ago

That's great. Thanks a lot!

Reply • Share ›

António Pereira • 4 years ago

Another great tutorial!
What is missing, from the point of view of a Python newbie, is some kind of introduction to concepts as virtualenv, and some more comments in the snippets since the code gets complex.

Reply • Share ›

Sam Lam • 4 years ago

hi I am using mac, I was trying to install those three web frameworks you mentioned, but it seems some something wrong when I ran the second command, I don't know if it's because I am running Python3.

Here is the error message:

virtualenv ~/envs/lsbaws

Using base prefix '/Applications/Canopy.app/appdata/canopy-1...

New python executable in /Users/jiahaolin/envs/lsbaws/bin/python

dyld: Library not loaded: @rpath/Python

Referenced from: /Users/jiahaolin/envs/lsbaws/bin/python

Reason: image not found

ERROR: The executable /Users/jiahaolin/envs/lsbaws/bin/python is not functioning

ERROR: It thinks sys.prefix is u'/Users/jiahaolin' (should be u'/Users/jiahaolin/envs/lsbaws')

ERROR: virtualenv is not compatible with this system or executable

Reply • Share ›

RichMurphy131 • 4 years ago

That was a wonderfully insightful and educational read. I'll be keeping my ear to the ground for Part 3 for sure!

Reply • Share ›

ImoApps • 4 years ago

Thank you!, wonderful tutorial, short ,sweet and right on the point .I will definitely implement this experiment .
thanks for the encouragement

Reply • Share ›

Ammar • 4 years ago

Great tutorial. Thanks for sharing with us..!!

Reply • Share ›

asdasdasd • 4 years ago

Even though I am quite familiar with this topic, I find your post an excellent summary.

Reply • Share ›

Wong Zigii • 4 years ago

I can't wait to see the next one !

Reply • Share ›

James Walker • 4 years ago

This is a brilliant tutorial series. Excited for what's next and the coming book! Keep up the great work - this content os very informative and very well explained but above all very useful!

Reply • Share ›

iKIsR • 4 years ago

Well worth the wait, keep these coming please.

• Reply • Share ›

psykrsna • 4 years ago

Thank you. Very informative and finely written.

• Reply • Share ›

John Stinson • 4 years ago

This is definitely helpful. Keep it up!

• Reply • Share ›

John Stinson ➚ John Stinson • 4 years ago

As for the question posed at the end, my guess would be to increase request_queue_size from one to something higher. From my look into the socket library, most set it to a five.

I have a question for you Ruslan. Are there other libraries (similar to how you used sockets) that could create this server?

• Reply • Share ›

Load more comments

Social

- github (https://github.com/rspivak/)
- twitter (https://twitter.com/alienoid)
- linkedin (https://linkedin.com/in/ruslanspivak/)

Popular posts

Let's Build A Web Server. Part 1. (https://ruslanspivak.com/lsbaws-part1/)

Let's Build A Simple Interpreter. Part 1. (https://ruslanspivak.com/lsbasi-part1/)

Let's Build A Web Server. Part 2. (https://ruslanspivak.com/lsbaws-part2/)

Let's Build A Web Server. Part 3. (https://ruslanspivak.com/lsbaws-part3/)

Let's Build A Simple Interpreter. Part 2. (https://ruslanspivak.com/lsbasi-part2/)

Disclaimer

Some of the links on this site have my Amazon referral id, which provides me with a small commission for each sale. Thank you for your support.

↑ Back to top