WSGI

Part 1, Chapter 2

Before we dive into the details of WSGI, let's look at what happens when a user uses a web application from a bird-eye's view.

Web Server

Let's look at the world through a web server's eyes...

Imagine for a moment that you are a web server, like Gunicorn. Your job consists of the following parts:

- You sit around and wait patiently for a request from some kind of a client.
- When a client comes to you with a request, you receive it.
- Then, you take this request to someone called PythonApp and say to him, "Hey dude, wake up! Here's a request from a very important client. Please, do something about it."
- You get a response from this PythonApp.
- You then deliver this response back to your client.

This is the only thing you do. You just serve your clients. You know nothing about the content or anything else. That's why you are so good at it. You can even scale up and down processing depending on the demand from the clients. You are focused on this single task.

Web App

PythonApp is your software. While a web server should exist and wait for an incoming request all the time, your software exists only at the execution time:

- The web server wakes it up and gives it the request.
- It takes the request and executes some commands on it.
- It returns a response to the web server.
- It goes back to sleep.
- The web server delivers this response back to his client.

The only thing it does is execute when asked to do so.

The Problem

The scenario above is all good. However, a web server's conversation with PythonApp could have gone a little differently.

Instead of:

"Hey dude, wake up! Here's a request from a very important client. Please, do something about it."

It could have been this:

"Эй, чувак, проснись! Вот запрос от очень важного клиента. Пожалуйста, сделай что нибудь."

Or this:

"Ehi amico, svegliati! Ecco una richiesta da un cliente molto importante. Si prega, fare qualcosa al riguardo"

Or even this:

"嘿，伙计，醒醒吧！这里是一个非常重要的客户端的请求。请做点什么"

Do you get it? The web server could have behaved in a number of different ways and PythonApp needs to learn all these languages to understand what it's saying and behave accordingly.

What this means is that in the past (before WSGI) you had to adapt your software to fit the requirements of a web server. Moreover, you had to write different kinds of wrappers in order to make it suitable across different web servers. Developers generally don't want to deal with such things. They just want to write code.

WSGI to the Rescue

Here is where WSGI comes in to play. You can think of it as a SET OF RULES for a web server and a web application.

The rules for a web server look like this:

Okay. If you want to talk to PythonApp, speak *these* words and sentences. Also, learn *these* words as well which it will speak back to you. Furthermore, if something goes wrong, *here* are the curse words that PythonApp will say and *here* is how you should react to them.

And the rules for a web application look like this:

Okay. If you want to talk to a web server, learn *these* words because a web server will be using them when addressing you. Also, you use *these* words and be sure that a web server understands them. Furthermore, if something goes wrong, use *these* curse words and behave in *this* way.

Example

Enough talk, let's fight!

Let's take a look at the WSGI application interface to see how it should behave. According to PEP 333, the document which specifies the details of WSGI, the application interface is implemented as a callable object such as a function, a method, a class, or an instance with a `__call__` method. This object should accept two positional arguments and return the response body as strings in an iterable.

The two arguments are:

- a dictionary with environment variables
- a callback function that will be used to send HTTP statuses and HTTP headers to the server

Now that we know the basics, let's create a web framework which will take away some market share from Django itself! Our web framework will do something that no one is doing right now: IT WILL PRINT ALL ENVIRONMENT VARIABLES IT RECEIVES. Genius!

Okay, open your text editor of choice, and create that callable object which receives two arguments:

```
def application(environ, start_response):
    pass
```

Easy enough. Now, let's prepare the response body that we want to return back to the server:

```
def application(environ, start_response):
    response_body = [
        f'{key}: {value}' for key, value in sorted(environ.items())
    ]
    response_body = '\n'.join(response_body)
```

Here, we are iterating over `environ.items()` and making a list of key/value pairs like so:

```
[
    "ENV_VAR_1_NAME": "ENV_VAR_1_VALUE",
    "ENV_VAR_2_NAME": "ENV_VAR_2_VALUE",
    ...
]
```

For example:

```
[
    "USER: Jahongir",
    "LANG: en_US.UTF-8"
]
```

After that, we are joining all the elements of the list `response_body` using `\n` as a line break.

Now, let's prepare the status and headers, and then call that callback function:

```
def application(environ, start_response):
    response_body = [
        f'{key}: {value}' for key, value in sorted(environ.items())
    ]
    response_body = '\n'.join(response_body)

    status = '200 OK'

    response_headers = [
        ('Content-type', 'text/plain'),
    ]

    start_response(status, response_headers)
```

And finally, let's return the response body in an iterable:

```
def application(environ, start_response):
    response_body = [
        f'{key}: {value}' for key, value in sorted(environ.items())
    ]
    response_body = '\n'.join(response_body)

    status = '200 OK'

    response_headers = [
        ('Content-type', 'text/plain'),
    ]

    start_response(status, response_headers)

    return [response_body.encode('utf-8')]
```

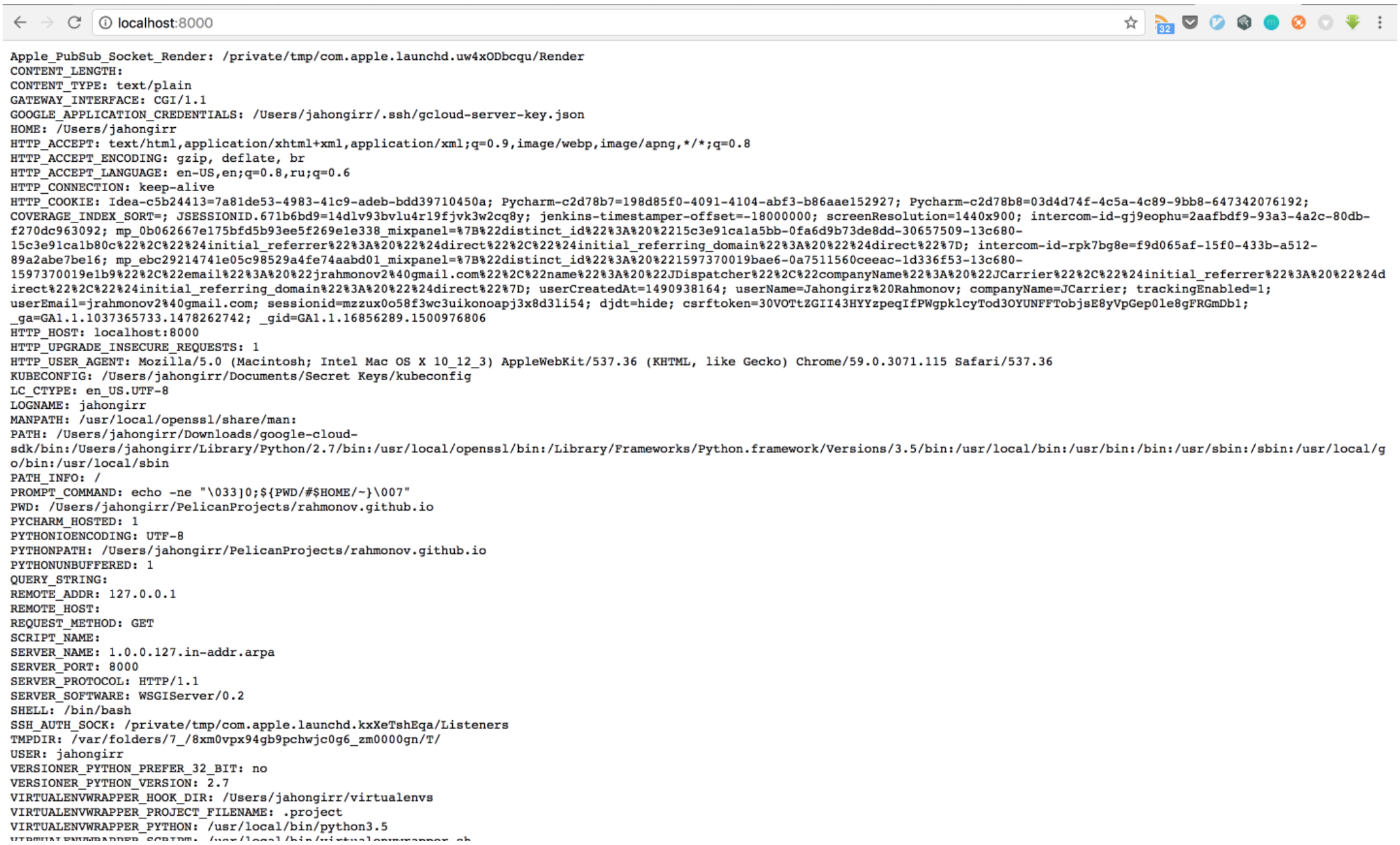
That's it. Our genius web framework is ready. Of course, we need a web server to serve our application and here we'll be using Python's bundled WSGI server.

> Want to learn more about the WSGI server interface? Take a look at it here.

Now, let's serve our application:

```python
from wsgiref.simple_server import make_server


def application(environ, start_response):
    response_body = [
        '{key}: {value}'.format(key=key, value=value) for key, value in sorted(environ.items())
    ]
    response_body = '\n'.join(response_body)

    status = '200 OK'

    response_headers = [
        ('Content-type', 'text/plain'),
    ]

    start_response(status, response_headers)

    return [response_body.encode('utf-8')]


server = make_server('localhost', 8000, app=application)
server.serve_forever()
```

Save this file as *wsgi_demo.py* and run it via *python wsgi_demo.py*. Then, go to localhost:8000 and you will see all the environment variables listed:



YES! This framework will be very popular!

Now that we know about the WSGI application interface, let's talk about something that we deliberately skipped earlier: Middleware.

Middleware

Middleware changes things a bit. With middleware, the above scenario will look like this:

- The web server gets a request.
- Now, instead of talking directly to PythonApp, it will send it through a postman (e.g., middleware).
- The postman delivers the request to PythonApp.
- After PythonApp does his job, it gives the response to the postman.
- The postman then delivers the response to the web server.

The only thing to note is that while the postman is delivering the request/response, it may tweak it a little bit.

Let's see it in action. We'll now write a middleware that reverses the response from our application:

```python
class Reverseware:
    def __init__(self, app):
        self.wrapped_app = app

    def __call__(self, environ, start_response, *args, **kwargs):
        wrapped_app_response = self.wrapped_app(environ, start_response)
        return [data[::-1] for data in wrapped_app_response]
```

Two things to note here:

1. First, web servers will talk to this middleware first and thus it must adhere to the same WSGI standards. That is, it should be a callable object that receives two params (`environ` and `start_response`) and then returns the response as an iterable.

2. Second, this middleware is getting the response from the app that it wraps and then it is tweaking it a little -- reversing the response, in this case. This is generally what middlewares are used for: tweaking the request and the response. We'll see a much more useful example later in this course.

Now, if we insert this code in the example above, the full code will look like this:

```
# wsgi_demo.py

from wsgiref.simple_server import make_server


class Reverseware:
    def __init__(self, app):
        self.wrapped_app = app

    def __call__(self, environ, start_response, *args, **kwargs):
        wrapped_app_response = self.wrapped_app(environ, start_response)
        return [data[::-1] for data in wrapped_app_response]


def application(environ, start_response):
    response_body = [
        f'{key}: {value}' for key, value in sorted(environ.items())
    ]
    response_body = '\n'.join(response_body)

    status = '200 OK'

    response_headers = [
        ('Content-type', 'text/plain'),
    ]

    start_response(status, response_headers)

    return [response_body.encode('utf-8')]


server = make_server('localhost', 8000, app=Reverseware(application))
server.serve_forever()
```

Now, if you run it, you will see something like this in your browser:


```
)0 ,1( :noisrev.igsw
ptth :emehcs_lru.igsw
eslaF :ecno_nur.igsw
eurT :daerhtitlum.igsw
eslaF :ssecorpitlum.igsw
>7=eman redaeRdereffuB.oi_< :tupni.igsw
>'repparWeliF.litu.ferigsw' ssalc< :repparw_elif.igsw
>'8-FTU'=gnidocne 'w'=edom '>rredts<'=eman repparWOItxeT.oi_< :srorre.igsw
4.3nohtyp/nib/4.3/snoisreV/krowemarf.nohtyP/skrowemarF/yrarbiL/ :__REHCNUAL_VNEVYP__
0x0:0x0:5F1x0 :GNIDOCNE_TXET_RESU_FC__
2151.mrahcyp.sniarbtej.moc :EMAN_ECIVRES_CPX
0x0 :SGALF_CPX
svnelautriv/rrignohaj/sresU/ :EMOH_NOKROW
1 :DC_NOKROW_REPPARWVNELAUTRIV
hs.repparwvnelautriv/nib/lacol/rsu/ :TPIRCS_REPPARWVNELAUTRIV
5.3nohtyp/nib/lacol/rsu/ :NOHTYP_REPPARWVNELAUTRIV
tcejorp. :EMANELIF_TCEJORP_REPPARWVNELAUTRIV
svnelautriv/rrignohaj/sresU/ :RID_KOOH_REPPARWVNELAUTRIV
7.2 :NOISREV_NOHTYP_RENOISREV
on :TIB_23_REFERP_NOHTYP_RENOISREV
rrignohaj :RESU
/T/ng0000mz_6g0cjwhcp9bg49xpv0mx8/_7/sredlof/rav/ :RIDPMT
srenetsiL/aqEhsTeXxk.dhcnual.elppa.moc/pmt/etavirp/ :KCOS_HTUA_HSS
hsab/nib/ :LLEHS
2.0/revreSIGSW :ERAWTFOS_REVRES
1.1/PTTH :LOCOTORP_REVRES
0008 :TROP_REVRES
apra.rdda-ni.721.0.0.1 :EMAN_REVRES
 :EMAN_TPIRCS
TEG :DOHTEM_TSEUQER
 :TSOH_ETOMER
1.0.0.721 :RDDA_ETOMER
 :GNIRTS_YREUQ
1 :DEREFFUBNUNOHTYP
oi.buhtig.vonomhar/stcejorPnacileP/rrignohaj/sresU/ :HTAPNOHTYP
8-FTU :GNIDOCNEOINOHTYP
1 :DETSOH_MRAHCYP
oi.buhtig.vonomhar/stcejorPnacileP/rrignohaj/sresU/ :DWP
"700\}~/EMOH$#/DWP{$;0]330\" en- ohce :DNAMMOC_TPMORP
/ :OFNI_HTAP
nibs/lacol/rsu/:nib/og/lacol/rsu/:nibs/:nibs/rsu/:nib/:nib/rsu/:nib/lacol/rsu/:nib/5.3/snoisreV/krowemarf.nohtyP/skrowemarF/yrarbiL/:nib/lssnepo/lacol/rsu/:nib/7.2/nohtyP/yrarbiL/rri
gnohaj/sresU/:nib/kds-duolc-elgoog/sdaolnwoD/rrignohaj/sresU/ :HTAP
:nam/erahs/lssnepo/lacol/rsu/ :HTAPNAM
rrignohaj :EMANGOL
8-FTU.SU_ne :EPYTC_CL
gifnocebuk/syeK terceS/stnemucoD/rrignohaj/sresU/ :GIFNOCEBUK
63.735/irafaS 511.1703.0.95/emorhC )okceG ekil ,LMTHK( 63.735/tiKbeWelppA )3_21_01 X SO caM letnI ;hsotnicaM( 0.5/allizoM :TNEGA_RESU_PTTH
1 :STSEUQER_ERUCESNI_EDARGPU_PTTH
0008:tsohlacol :TSOH_PTTH
6086790051.98265861.1.1AG=dig_ ;2472628741.3375637301.1.1AG=ag_ ;1bDmGRFg8el0peGpVy8EsjboTFFNUYO3doTyclkpgWPfIqepzYYH34IIGZtTOV03=nekotfrsc ;edih=tdjd
;45il3d8x3jpaonokiu3cw3f85o0xuzzm=dinoisses ;moc.liamg04%2vonomharj=liamEresu ;1=delbanEgnikcart ;reirraCJ=emaNynapmoc ;vonomhaR02%zrignohaJ=emaNresu ;4618390941=tAdetaerCresu
;D7%22%tcerid42%22%02%A3%22%niamod_gnirrefer_laitini42%22%C2%22%tcerid42%22%02%A3%22%rerrefer_laitini42%22%C2%22%reirraCJ22%02%A3%22%emaNynapmoc22%C2%22%rehctapsiDJ22%02%A3%22%eman22
%C2%22%moc.liamg04%2vonomharj22%02%A3%22%liame22%C2%22%9b1e9100737951-086c31-35f633d1-caeec0651157a0-
```

Beautiful!

Conclusion

In this chapter, you learned what WSGI is and why it is needed. You also built a small, dummy framework (if you can even call it that) that simply receives any request and returns its environment variables in reverse as a response.

Alright, that's it for this chapter. If you want to learn more about WSGI, please see the updated PEP 3333.

See you in chapter two. Adios!