WSGI, une technique pour des applications Web en Python

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 16 mars 2009. Dernière mise à jour le 11 juin 2011

http://www.bortzmeyer.org/wsgi.html

Pour créer une application Web en Python, il existe plusieurs solutions. La plus ancienne est d'utiliser le standard CGI qui permet de créer très simplement des applications Web sommaires. La plus à la mode est d'installer un des innombrables environnements de développement et d'exécution, comme Django. Il existe encore une autre solution, développer des applications en WSGI.

D'abord, qu'est-ce qui ne va pas avec les CGI ? Cette méthode, la plus ancienne du Web, est ultra-simple (n'importe quel programmeur, n'ayant jamais fait de Web, peut écrire un CGI en cinq minutes, et ceci quel que soit le langage de programmation utilisé), robuste, ne nécessite pas d'installer quoi que ce soit de particulier, fonctionne avec tous les serveurs HTTP. En Python, l'excellent module cgi <http://docs.python.org/library/cgi.html> de la bibliothèque standard rend cela encore plus facile.

Mais les CGI ont des défauts, liés au fait que tout (machine virtuelle Python, connexion aux bases de données, etc) est initialisé à chaque requête HTTP. Les CGI sont donc en général lents. Mais ils restent imbattables pour les petites applications développées vite fait.

Les développeurs Python modernes vont alors crier en anglais "*framework !*" (environnement de développement et d'exécution). Ces environnements, mis à la mode par Ruby on Rails, sont innombrables dans le monde Python (le premier ayant été Zope, le plus populaire aujourd'hui étant Django). On en compte des dizaines activement maintenus et le nombre augmente tous les jours. Le simple choix d'un environnement peut prendre des jours, l'installation et l'administration système sont loin d'être simples et la pérennité des applications développées dans un environnement est loin d'être assurée, puisque les applications sont dépendantes d'un environnement. En outre, certains environnements imposent (ou, en tout cas, poussent fortement) à utiliser des choses dont je me méfie comme les ORM. Enfin, personnellement, j'ai du mal à me plonger dans ces usines à gaz complexes, qui utilise chacune son vocabulaire particulier. Je ne suis pas développeur d'applications Web à temps plein donc le temps que je passerai à apprendre un gros machin comme Django ne pourra pas être amorti sur de nombreux projets.

Pour une petite (toute petite) application Python sur un site Web, le moteur de recherche <http://www.bortzmeyer.org/moteur-recherche-wsgi.html> de ce blog, je cherchais s'il n'y avait

pas mieux ou plus récent dans la catégorie « trucs ultra-simple pour le programmeur, qui permet de faire une application Web en trente secondes sans se prendre la tête ». Je cherchais à retrouver la simplicité et la rapidité de développement des CGI avec un truc qui aie moins de problèmes de performance et de passage à l'échelle.

Il existe plusieurs solutions dans cette catégorie. Par exemple, mod_python est un module Apache qui permet d'embarquer la machine virtuelle Python dans le serveur HTTP Apache et donc de démarrer bien plus rapidement. Porter une application CGI en mod_python n'est pas trivial (le mode de fonctionnement est assez différent) et mod_python semble moins maintenu de nos jours (mais il reste la meilleure solution lorsqu'on veut accéder aux fonctions d'Apache, pour modifier le comportement de ce serveur). En outre, mod_python est spécifique à Apache.

Une autre solution envisageable est WSGI. Décrit dans le PEP 333 <http://www.python.org/dev/peps/pep-0333/>, indépendant du serveur HTTP, WSGI est un standard d'interface entre le serveur HTTP et Python. Il est plutôt prévu pour fournir aux développeurs d'environnements une interface sur laquelle s'appuyer, afin de garantir que leur environnement tournera sur tous les serveurs HTTP. WSGI n'est typiquement jamais promu comme interface pour le développeur d'applications finales.

C'est bien dommage, car WSGI a plusieurs propriétés intéressantes :

— Rapide (pour mon application, qui utilise un SGBD, mesuré avec echoping <http://echoping.sourceforge.net/>, le gain de performance est net, la médiane passe de 0,4 s (CGI) à 0,1 s (WSGI), sur la machine locale).
— Une mise en œuvre de WSGI pour Apache, mod_wsgi <http://www.modwsgi.org>, a un mode « démon » où les processus qui exécutent le code Python ne sont **pas** dans le serveur HTTP mais sont des démons séparés, comme avec FastCGI. Ainsi, une bogue dans le code Python (par exemple une fuite de mémoire) n'affectera pas le processus Apache (l'un des problèmes récurrents de mod_python ou de mod_perl).
— Modèle de programmation suffisamment simple pour que les applications CGI puissent être portées très vite.

(Et quelles sont les différences avec FastCGI ? Dans les deux cas, le serveur HTTP envoie la requête à un processus externe qui tourne en permanence, ce qui évite de payer le coût du démarrage pour chaque requête. Mais les deux solutions sont distinctes : FastCGI spécifie le protocole entre le serveur HTTP et le serveur d'application. WSGI normalise l'API utilisée par le serveur HTTP pour parler au serveur d'application - et celle utilisée par le serveur d'application pour parler à l'application. WSGI est donc spécifique à un langage - Python - mais de plus haut niveau. On peut donc implémenter WSGI sur FastCGI.)

À quoi ressemble une application WSGI ? Un exemple trivial, qui affiche les variables d'environnement est :

```
import os

def application(environ, start_response):
    status = '200 OK'
    output = str(environ.keys()) + "\n"
    response_headers = [('Content-Type', 'text/plain'),
                        ('Content-Length', str(len(output)))]
    start_response(status, response_headers)

    return [output]
```

On y voit les points importants : WSGI appelle un sous-programme nommé `application`, il lui passe un objet qui contient les variables d'environnement, ici nommé `environ` et une fonction à appeler avec le code de retour HTTP (ici 200) et les en-têtes HTTP (comme le type MIME `Content-Type`).

Pour comparer, le programme qui affiche les variables d'environnement ressemblerait à ceci en CGI :

```
import os

print "Content-Type: text/plain"
print ""
print os.environ.keys()
```

Bien sûr, WSGI est beaucoup plus riche que cela et on peut faire bien d'autre choses. Par exemple, pour répartir les requêtes selon l'URI indiqué, on regarde la variable `PATH_INFO` et cela donne :

```
def application(environ, start_response):
    if environ['PATH_INFO'] == "/toto":
        return toto(start_response)
    if environ['PATH_INFO'] == "/shadok":
        return shadok(start_response)
    else:
        return default(start_response, environ['PATH_INFO'])
...
def shadok(start_response):
    status = '200 OK'
    output = "<h1>Ga Bu Zo Meu</h1>\n"
    response_headers = [('Content-type', 'text/html'),
                        ('Content-Length', str(len(output)))]
    start_response(status, response_headers)
    return [output]
...
[Idem pour les autres fonctions]
```

Ainsi, on peut bâtir des applications arbitrairement complexes. Aujourd'hui, la plupart des environnements de développement et d'exécution Web pour Python sont développés en utilisant WSGI.

Sans aller jusque là, un exemple réel est le moteur de recherche de ce blog `<http://www.bortzmeyer.org/moteur-recherche-wsgi.html>`.

En pratique, comment installe t-on des WSGI sur Apache ? (Mais rappelez-vous qu'un gros avantage de WSGI est la capacité à fonctionner sur tous les serveurs HTTP.) Il faut installer le logiciel (sur Debian, `aptitude install libapache2-mod-wsgi`, sur Gentoo `emerge mod_wsgi`, etc). On configure ensuite Apache, par exemple ainsi :

```
WSGIScriptAlias /search /var/www/www.bortzmeyer.org/wsgis/search.py
WSGIDaemonProcess bortzmeyer.org processes=3 threads=10 display-name=%{GROUP}
WSGIProcessGroup bortzmeyer.org
<Directory /var/www/www.bortzmeyer.org/wsgis>
        Options -Indexes -Multiviews +ExecCGI
        Order allow,deny
        Allow from all
</Directory>
```

Cette configuration fait fonctionner WSGI en mode démon, avec trois démons.

Du fait que le démon tourne en permanence, ses variables sont conservées d'une requête sur l'autre. Par exemple, voici une démonstration </apps/counter> qui incrémente un compteur et affiche le PID et l'adresse IP d'origine. Le PID indique le démon qui a exécuté le programme et le compteur s'incrémente à chaque requête. Attention, il n'y a pas de mémoire partagée, chaque démon a son compteur. Voici le code :

```
# Ces variables seront initialisées au lancement du démon
# et tarderont leur valeur d'une requête HTTP sur l'autre.
calls = 0
pid = os.getpid()
...
def counter(start_response, client):
    status = '200 OK'

    output = """
    <html><head><title>WSGI works</title></head>
    <body>
    <h1>WSGI works</h1>
    <p>You are the %i th visitor for this WSGI daemon (Process ID %i).
    Reload the page to see the changes.</p>
    <p>You arrived from machine %s.</p>
    </body>
    </html>""" % (calls, pid, client)

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

    return [output]
...
def application(environ, start_response):
    global calls
    calls += 1
    if environ['PATH_INFO'] == "/counter":
        return counter(start_response, environ['REMOTE_ADDR'])
    else:
        return default(start_response, environ['PATH_INFO'])
```

Dans les exemples ci-dessus, le code WSGI doit gérer tout un tas de problèmes de bas niveau, comme d'envoyer correctement des en-têtes HTTP. Le programmeur paresseux pourra aussi faire du WSGI en utilisant des bibliothèques comme webob <http://pythonpaste.org/webob/> ou wsgiref <http://docs.python.org/library/wsgiref.html>. Je ne les ai pas encore testées mais cela peut simplifier le développement.

Puisqu'on parle de simplifier le développement, existe t-il un moyen plus simple de tester ses applications WSGI en local, sans avoir à installer un Apache sur sa machine ? Oui, on peut utiliser, là encore, la bibliothèque wsgiref, elle contient un serveur HTTP minimum, très pratique pour les tests locaux. Supposons que l'application « compteur » citée plus haut soit dans un fichier myapps.py. Le programme Python suivant va l'importer, puis lancer un serveur HTTP sur le port indiqué :

```
import wsgiref.simple_server as server
import myapps

port = 8080

httpd = server.make_server('', port, myapps.application)
print "Serving HTTP on port %i..." % port
# Respond to requests until process is killed
httpd.serve_forever()
```

En regardant avec un navigateur `http://localhost:8080/counter`, on appelera l'application qu'on veut déboguer, avec le même contexte (puisque WSGI est une norme, il n'est pas spécifique à Apache, il marchera aussi bien, avec `simple_server`).

Enfin, un autre bon truc pour déboguer des applications WSGI en local, sans passer du tout par un serveur HTTP. Écrire ce code dans le programme principal :

```
# Si et seulement si ce code est lancé interactivement
if __name__ == '__main__':
    def local_debug(status, headers):
        print status
        print headers
    import os
    print "".join(application(os.environ, local_debug))
```

Le script WSGI sera alors exécuté et affichera ses résultats.

Les idées exposées dans cet article viennent en bonne partie de "*Writing Blazing Fast, Infinitely Scalable, Pure-WSGI Utilities*" <http://www.eflorenzano.com/blog/post/writing-blazing-fast-infinitely-s
>.