Getting Started with WSGI

written on Monday, May 21, 2007

I finally finished the written matura and have some more time to work on projects and write articles. One of the things I wanted to write for a long time is a WSGI tutorial that does not require a specific framework or implementation. So here we go.

What's WSGI?

Basically WSGI is lower level than CGI which you probably know. But in difference to CGI, WSGI does scale and can work in both multithreaded and multi process environments because it's a specification that doesn't mind how it's implemented. In fact WSGI is not CGI because it's between your web application and the webserver layer which can be CGI, mod_python, FastCGI or a webserver that implements WSGI in the core like the python stdlib standalone WSGI server called wsgiref.

WSGI is specified in the PEP 333 and adapted by various frameworks including the well known frameworks django and pylons.

If you are too lazy to read the pep 333 here's a short summary:

- WSGI application are callable python objects (functions or classes with a __*call*__ method that are passed two arguments: a WSGI environment as first argument and a function that starts the response.
- the application has to start a response using the function provided and return an iterable where each yielded item means writing and flushing.
- The WSGI environment is like a CGI environment just with some additional keys that are either provided by the server or a middleware.
- you can add middlewares to your application by wrapping it.

Because that's a lot of information let's ignore it for now and have a look at a basic WSGI application:

Extended Hello World

Here a simple, but not too simple example of a WSGI application that says *Hello World!* where World can be specified via url parameter.

```
from cgi import parse_qs, escape

def hello_world(environ, start_response):
    parameters = parse_qs(environ.get('QUERY_STRING', ''))
    if 'subject' in parameters:
        subject = escape(parameters['subject'][0])
    else:
        subject = 'World'
    start_response('200 OK', [('Content-Type', 'text/html')])
    return ['''Hello %(subject)s
    Hello %(subject)s!

''' % {'subject': subject}]
```

As you can see the *start_response* function takes two arguments. A status string and a list of tuples that represent the response headers. What you cannot see because it's not used here and nowhere else is that the *start_response* function returns something. It returns a *write* function that directly writes to the webserver output stream. Because it bypasses middlewares (we'll cover that later) it's a terrible bad idea to use that function. For debugging purposes however it can be useful.

But how to start that application now? A webserver doesn't know how to handle that and neither does python because nothing calls that function. Because we're lazy we don't setup a server with WSGI support now but use the *wsgiref* WSGI standalone server bundled with python2.5 and higher. (You can also download it for python2.3 or 2.4)

Just add this to your file:

```
if __name__ == '__main__':
    from wsgiref.simple_server import make_server
    srv = make_server('localhost', 8080, hello_world)
    srv.serve_forever()
```

When you now start the file you should be able to get a *Hello John!* on *http://localhost:8080/?subject=John*.

Path Dispatching

You probably worked with CGI or PHP before. If you did so you know that you most of the time have multiple public files (*.pl* / *.php*) a user can access and that do something. Not so in WSGI. There you only have one file which consumes all paths. Thus if you have your server from the previous example still running you should get the same content on *http://localhost:8080/foo?subject=John*.

The accessed path is saved in the *PATH_INFO* variable in the WSGI environment, the real path to the application in *SCRIPT_NAME*. In case of the development server *SCRIPT_NAME* will be empty, but if you have a wiki that is mounted on *http://example.com/wiki* the *SCRIPT_NAME* variable would be */wiki*. This information can now be used to serve multiple indepentent pages with nice URLs.

In this example we have a bunch of regular expressions and match the current request against that:

```
import re
from cgi import escape

def index(environ, start_response):
    """This function will be mounted on "/" and display a link
    to the hello world page."""
    start_response('200 OK', [('Content-Type', 'text/html')])
    return ['''Hello World Application
               This is the Hello World application:

`continue <hello/>`_

''']

def hello(environ, start_response):
    """Like the example above, but it uses the name specified in the
URL."""
    # get the name from the url if it was specified there.
    args = environ['myapp.url_args']
    if args:
        subject = escape(args[0])
    else:
        subject = 'World'
    start_response('200 OK', [('Content-Type', 'text/html')])
    return ['''Hello %(subject)s
            Hello %(subject)s!

''' % {'subject': subject}]

def not_found(environ, start_response):
    """Called if no URL matches."""
    start_response('404 NOT FOUND', [('Content-Type', 'text/plain')])
    return ['Not Found']

# map urls to functions
urls = [
    (r'^$', index),
    (r'hello/?$', hello),
    (r'hello/(.+)$', hello)
]

def application(environ, start_response):
    """
    The main WSGI application. Dispatch the current request to
    the functions from above and store the regular expression
    captures in the WSGI environment as  `myapp.url_args` so that
    the functions from above can access the url placeholders.
    If nothing matches call the `not_found` function.
    """
    path = environ.get('PATH_INFO', '').lstrip('/')
    for regex, callback in urls:
        match = re.search(regex, path)
        if match is not None:
            environ['myapp.url_args'] = match.groups()
            return callback(environ, start_response)
    return not_found(environ, start_response)
```

Now that's a bunch of code. But you should get the idea how URL dispatching works. Basically if you now visit *http://localhost:8080/hello/John* you should get the same as above but with a nicer URL and a error 404 page if you enter the wrong url. Now you could improve that further by encapsulating *environ* in a request object and replacing the *start_response* call and the return iterator with a response objects. This is also what WSGI libraries like Werkzeug and Paste do.

By adding something to the environment we did something normally middlewares do. So let's try to write one that catches exceptions and renders them in the browser:

```
# import the helper functions we need to get and render tracebacks
from sys import exc_info
from traceback import format_tb

class ExceptionMiddleware(object):
    """The middleware we use."""

    def __init__(self, app):
        self.app = app

    def __call__(self, environ, start_response):
        """Call the application can catch exceptions."""
        appiter = None
        # just call the application and send the output back
        # unchanged but catch exceptions
        try:
            appiter = self.app(environ, start_response)
            for item in appiter:
                yield item
        # if an exception occours we get the exception information
        # and prepare a traceback we can render
        except:
            e_type, e_value, tb = exc_info()
            traceback = ['Traceback (most recent call last):']
            traceback += format_tb(tb)
            traceback.append('%s: %s' % (e_type.__name__, e_value))
            # we might have not a stated response by now. try
            # to start one with the status code 500 or ignore an
            # raised exception if the application already started one.
            try:
                start_response('500 INTERNAL SERVER ERROR', [
                               ('Content-Type', 'text/plain')])
            except:
                pass
            yield '\n'.join(traceback)

        # wsgi applications might have a close function. If it exists
        # it *must* be called.
        if hasattr(appiter, 'close'):
            appiter.close()
```

So how can we use that middleware now? If our WSGI application is called *application* like in the previous example all we have to do is to wrap it:

```
application = ExceptionMiddleware(application)
```

Now all occouring exceptions will be catched and displayed in the browser. Of course you don't have to do that because there are many libraries that do exactly that and with more features.

Deployment

Now where the application is "finished" it must be installed on the production server somehow. You can of course use wsgiref behind mod_proxy but there are also more sophisticated solutions available. Many people for example prefer using WSGI applications on top of FastCGI. If you have flup installed all you have to do is to defined a *myapplication.fcgi* with this code in:

```
#!/usr/bin/python
from flup.server.fcgi import WSGIServer
from myapplication import application
WSGIServer(application).run()
```

The apache config then could look like this:

```
<ServerName www.example.com>
    Alias /public /path/to/the/static/files
    ScriptAlias / /path/to/myapplication.fcgi/
</ServerName>
```

As you can see there is also a clause for static files. If you are in development mode and want to serve static files in your WSGI application there are a couple of middlewares (werkzeug and paste as well as "static" from Luke Arno's tools provide that) available.

NIH / DRY

Avoid the "Not Invented Here" problem and don't repeat yourself. Use the libraries that exist and their utilities! But there are so many! Which one to use? Here my suggestions:

Frameworks

Since Ruby on Rails appeared on the web everybody is talking about frameworks. Python has two major ones too. One that abstracts stuff very much and is called Django and the other that is much nearer to WSGI and called pylons. Django is an awesome framework but only as long as you don't want to distribute your application. It's if you have to create a webpage in no time. Pylons on the other hand requires more developer interaction and your applications are a lot easier to deploy.

There are other frameworks too but **my** experiences with them are quite bad or the community is too small.

Utility Libraries

For many situations you don't want a full blown framework. Either because it's too big for your application or your application is too complex that you can solve it with a framework. (You can solve any application with a framework but it could be that the way you have to solve it is a lot more complex than without the "help" of the framework)

For that some utility libraries exist:

- Paste — used by pylons behind the scenes. Implements request and response objects. Ships many middlewares.
- Werkzeug — minimal WSGI library we wrote for pocoo. Ships unicode away request and response objects as well as an advanced URL mapper and a interactive debugger.
- Luke Arno's WSGI helpers — various WSGI helpers in independent modules by Luke Arno.

There are also many middlewares out there. Just look for them at the Cheeseshop.

Template Engines

Here a list of template engines I often use and recommend:

- Genshi — the world's best XML template engine. But quite slow, so if you need a really good performance you have to go with something else.
- Mako — stupidely fast text based template engine. It's a mix of ERB, Mason and django templates.
- Jinja2 — sandboxed, designer friendly and quite fast, text based template engine. Of course my personal choice :D

Conclusion

WSGI rocks. You can simply create your own personal stack. If you think it's too complicated have a look at werkzeug and paste, they make things a lot easier without limiting you.

I hope this article was useful.

This entry was tagged python and wsgi

© Copyright 2015 by Armin Ronacher.

Content licensed under the Creative Commons attribution-noncommercial-sharealike License.
Contact me via mail, twitter, github or bitbucket. Tip me via gittip.
More info: imprint. Subscribe to Atom feed (or RSS)