Special Topic

WSGI and Web Services

Introduction

- Suppose you want to make parts of your program accessible via HTTP
- Example: A REST-based service

Disclaimer

- This is a huge topic
- Assume that you are already generally familiar with underlying core concepts
  - HTTP protocol
  - URIs
  - Data encodings: XML, JSON, CSV, etc.

Big Picture

- Your service will live at some URI

  http://mysite.com/myservice?parm=value&parm=value

- It will recognize various actions

  GET, POST, PUT, UPDATE, DELETE

- Data exchanged in standard formats

  text/plain
  text/csv
  application/xml
  application/json

WSGI

- Web Services Gateway Interface (WSGI)
- This is a standardized interface for creating Python web services
- Allows one to create code that can run under a wide variety of web servers and frameworks as long as they also support WSGI (and most do)
- So, what is WSGI?

WSGI in a Nutshell

- It's a programming specification for writing Python functions that receive and respond to HTTP requests
- Intentionally minimal (depends on no libraries, is not tied to any framework)
- Loosely originates from CGI programming (the mainstay of web programming in the 90s).

WSGI Example

- You just write simple functions like this

```
def hello_app(environ, start_response):
    status = "200 OK"
    response_headers = [ ('Content-type','text/plain')]
    response = []

    start_response(status,response_headers)
    response.append("Hello World\n")
    return response
```

WSGI Example

- Or alternatively, a class with a __call__ method

```
class HelloApp(object):
    def __init__(self):
        # Set up internal state
        self.nhellos = 0
    def __call__(self,environ,start_response):
        status = "200 OK"
        response_headers = [ ('Content-type','text/plain')]
        response = []

        self.nhellos += 1
        start_response(status,response_headers)
        response.append("Hello World\n")
        response.append("Said %d times\n" % self.nhellos)
        return response
```

- Use a class if you need to keep internal state

WSGI Applications

- Applications always receive just two inputs

```
def hello_app(environ, start_response):
    status = "200 OK"
    response_headers = [ ('Content-type','text/plain')]
    response = []

    start_response(status,response_headers)
    response.append("Hello World\n")
    return response
```

- environ - A dictionary of input parameters
- start_response - A callable (e.g., function)

WSGI Environment

- The environment contains CGI variables

```
def hello_app(environ, start_response):
    status = "200 OK"
    response_headers = [ ('Content-type','text/plain')]
```

```
environ['REQUEST_METHOD']
environ['SCRIPT_NAME']
environ['PATH_INFO']
environ['QUERY_STRING']
environ['CONTENT_TYPE']
environ['CONTENT_LENGTH']
environ['SERVER_NAME']
... (more not shown)
```

- Various information from the HTTP request

WSGI Environment

- Environment also contains some WSGI variables

```
def hello_app(environ, start_response):
    status = "200 OK"
    response_headers = [ ('Content-type','text/plain')]
```

```
environ['wsgi.input']
environ['wsgi.errors']
environ['wsgi.url_scheme']
environ['wsgi.multithread']
environ['wsgi.multiprocess']
...
```

- wsgi.input - A file-like object for reading data
- wsgi.errors - File-like object for error output

Processing Parameters

- Requests often have passed parameters

  http://mysite.com/myservice?**foo=a&bar=b**

- Here's an example of parsing (ugly)

```
import cgi
def sample_app(environ,start_response):
    fields = cgi.FieldStorage(environ['wsgi.input'],
                              environ=environ)
    # fields now has the params
    ...
    # Get various field values
    foo = form.getvalue('foo')
    bar = form.getvalue('bar')
```

- Will see a better way shortly

WSGI Responses

- To initiate a response, use the passed start_response function

```
def hello_app(environ, start_response):
    status = "200 OK"
    response_headers = [ ('Content-type','text/plain')]
    response = []

    start_response(status,response_headers)
    response.append("Hello World\n")
    return response
```

- You pass it two parameters
  - A status string (e.g., "200 OK")
  - A list of (header, value) pairs

WSGI Responses

- start_response() is a hook back to the server
- Gives the server information for formulating the response (status, headers, etc.)
- Prepares the server for receiving content data

WSGI Content

- Content is returned <u>as a sequence of byte strings</u>

```
def hello_app(environ, start_response):
    status = "200 OK"
    response_headers = [ ('Content-type','text/plain')]
    response = []

    start_response(status,response_headers)
    response.append("Hello World\n")
    return response
```

- Note: It is often a list of string fragments (if response is built in pieces).

WSGI Content Encoding

- WSGI applications must always produce bytes
- If working with Unicode, it must be encoded

```
def hello_app(environ, start_response):
    status = "200 OK"
    response_headers = [ ('Content-type','text/html')]

    start_response(status,response_headers)
    return [u"That's a spicy Jalape\u00f1o".encode('utf-8')]
```

- Be aware that Unicode can sneak in even when you're not expecting it. Best to plan for it.

WSGI Deployment

- The main point of WSGI is to simplify deployment of web applications
- You will notice that the interface depends on no third party libraries, no objects, or even any standard library modules
- That is intentional. WSGI apps are supposed to be small self-contained units that can plug into other environments

A Simple WSGI Server

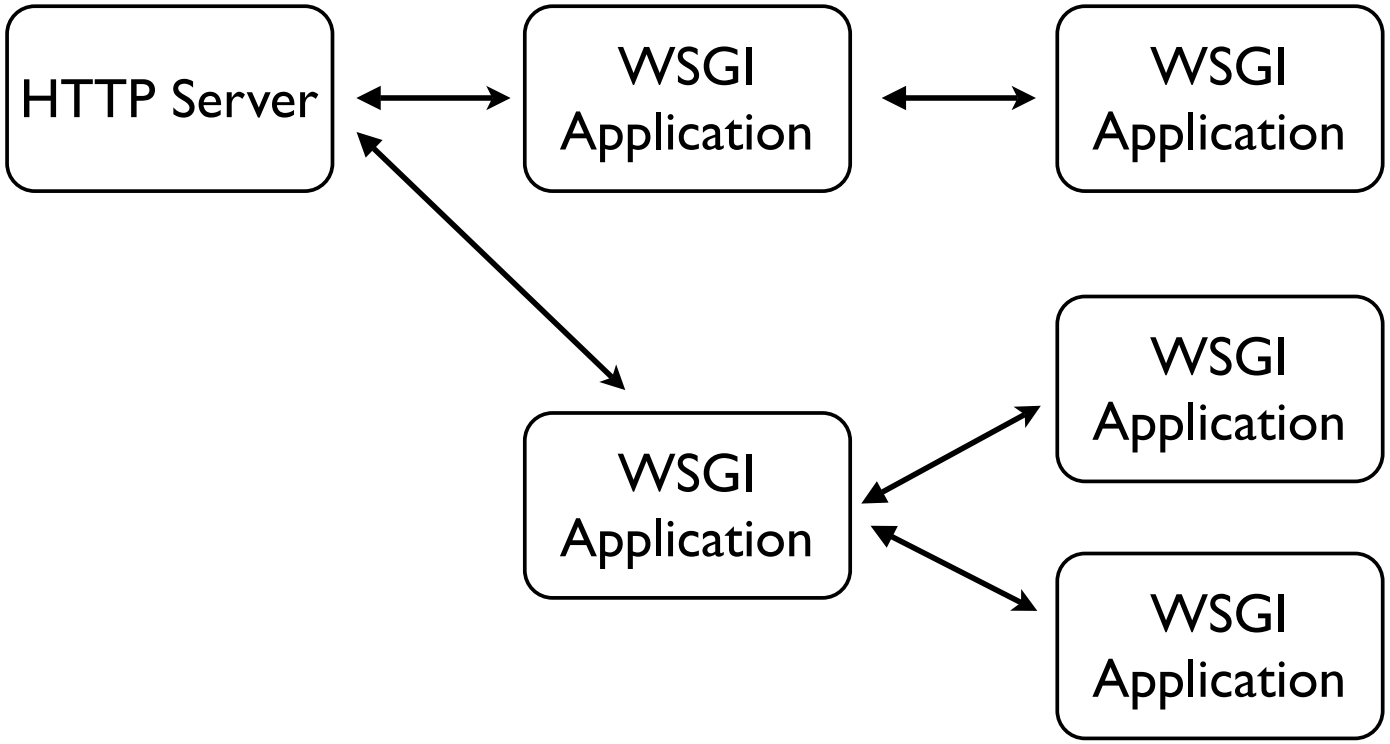
- Running a simple stand-alone WSGI server

```
from wsgiref import simple_server
httpd = simple_server.make_server("",8080,hello_app)
httpd.serve_forever()
```

- This runs an HTTP server for testing
- You probably wouldn't deploy anything using this, but if you're developing code on your own machine, it can be useful

WSGI Big Picture

- WSGI can be used as a mechanism for gluing different web software components together



WSGI Tools

- There are many tools to simplify development
- Again a big topic
- Will discuss two of them
    - WebOb
    - Paste

WebOb

http://webob.org/

- A small library that puts a higher-level interface around WSGI requests and responses
- Hides a lot of gory HTTP/WSGI details
- Also simplifies things such as testing

WebOb

- Skeleton example

```
from webob import Request, Response

def simple_app(environ, start_response):
    req = Request(environ)
    # Do things with req
    ...
    res = Response()
    res.body = "Hello World\n"
    return res(environ, start_response)
```

- It wraps low-level WSGI details with a higher-level interface

WebOb Requests

- Represents an incoming request

```
req = Request(environ)

# Attributes
req.method
req.path
req.path_info
req.content_type
req.body
req.remote_user
req.remote_addr
req.query_string
req.params
req.cookies
...
```

WebOb Responses

- Represents a response

```
res = Response()

# Attributes
res.status
res.content_type
res.charset
res.body                    (binary)
res.text                    (unicode)
res.headers
...
```

- Many more (caching, cookies, etc.)

WebOb Responses

- There are pre-built responses for various HTTP response and error codes

```
from webob.exc import *

res = HTTPNotFound()
res = HTTPServerError()
res = HTTPUnauthorized()
...
```

- Just create and use as the response

WebOb Testing

- Requests can be instantiated manually
- You can run them on WSGI apps

```
req = Request.blank("/subscribe")
req.query_string = "name=Dave&email=dave@dabeaz.com"
req.method="POST"
...
res = req.get_response(some_app)   # Call an WSGI app

# Look at the response
...
```

- Nice feature : Can experiment with your code without having to run it in a web server

Paste

http://pythonpaste.org/

- A collection of minimalistic tools for running and deploying WSGI applications
- It's something you might use if using a full-fledged web framework is overkill
- Will show just a few examples

Paste HTTP Server

- Running a standalone HTTP server on a WSGI application

```
def wsgi_app(environ, start_response):
    ...

if __name__ == '__main__':
    from paste import httpserver
    httpserver.server(wsgi_app,post='8080')
```

Paste URL Mapping

- Mapping URLs to different WSGI apps

```
def hello_app(environ, start_response):
    ...

def foo_app(environ, start_response):
    ...

from paste.urlmap import URLMap
mapper = URLMap()
mapper['/hello'] = hello_app
mapper['/services/foo'] = foo_app

if __name__ == '__main__':
    from paste import httpserver
    httpserver.server(mapper,post='8080')
```

Final Comments

- WSGI is intentionally meant to be minimal
- You can use it directly to make specialized web services and other applications
- An alternative to trying to work with a large web framework
- In theory, WSGI should integrate with a wide variety of Python-related web packages

Sample Code

- Look in:

  PythonClass/Solutions/wsgi/

- Simple examples of WSGI and WebOb