WSGI from Start to Finish

EuroPython 2010.

Gustavo Narea

Who's speaking

- ¡Hola!
- Web Application Developer.
- Contributes to WSGI projects.

Goals

- Explain what your framework does under-the-hood.
- More efficient troubleshooting.
- Integrate third party libraries and applications.
- Write framework-independent libraries and applications.
- Learn about existing WSGI-based software.

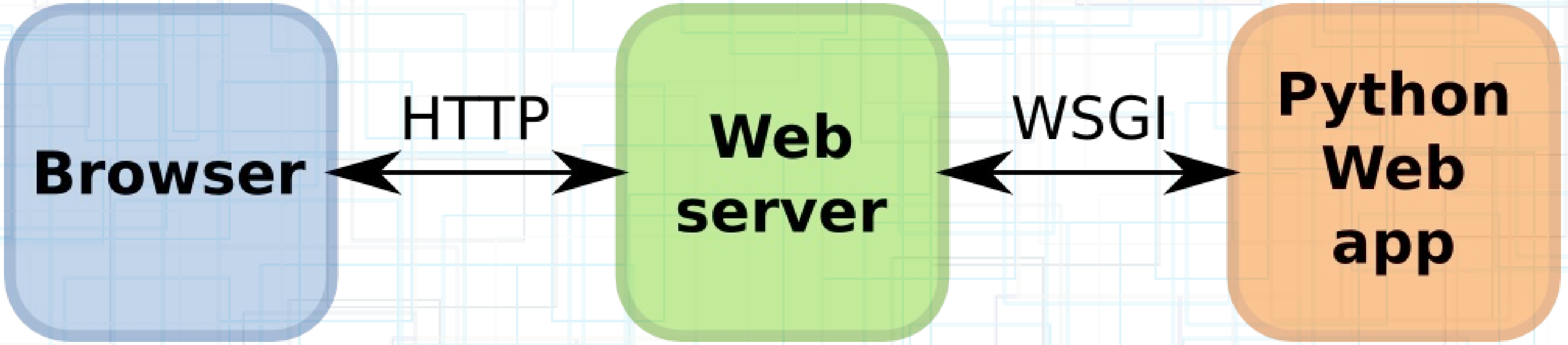
Updates after the tutorial

- This presentation was modified to refer to working examples and fix errata.
- You probably downloaded this presentation with the examples. If not, go to *gustavonarea.net/talks/* to get them.
- Read the instructions on how to install some of them.
- They are not essential to understand the presentation.

The big picture

Introduction

What's WSGI?



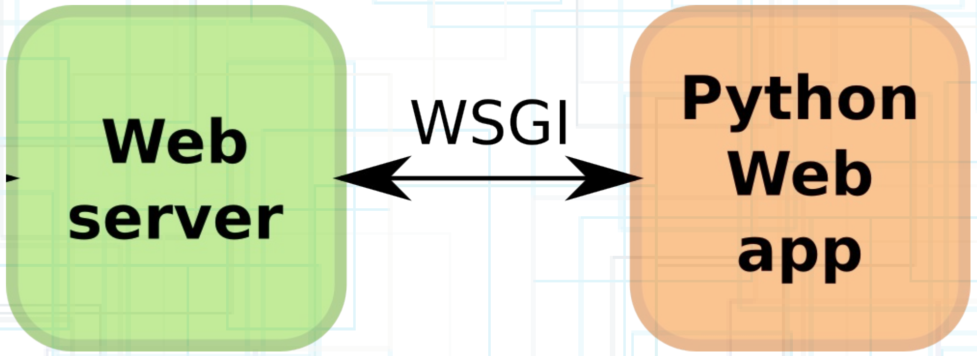
HTTP = HyperText Transfer Protocol
WSGI = Web Server Gateway Interface

Key facts about WSGI

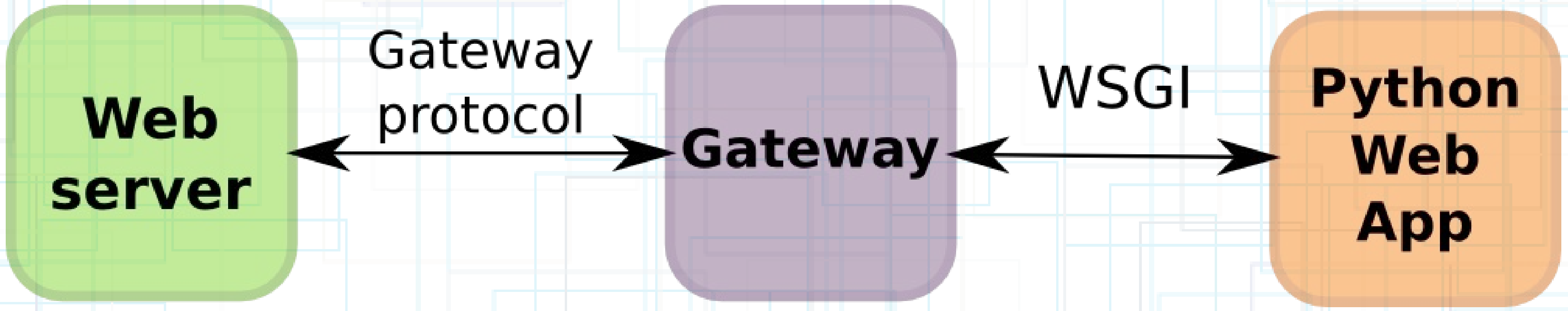
- Python “Standard” (PEP-333).
- Created in 2003.
- Inspired by CGI.
- Officially supported by all the popular frameworks.
- Applications can run on virtually any HTTP server.

Servers and gateways

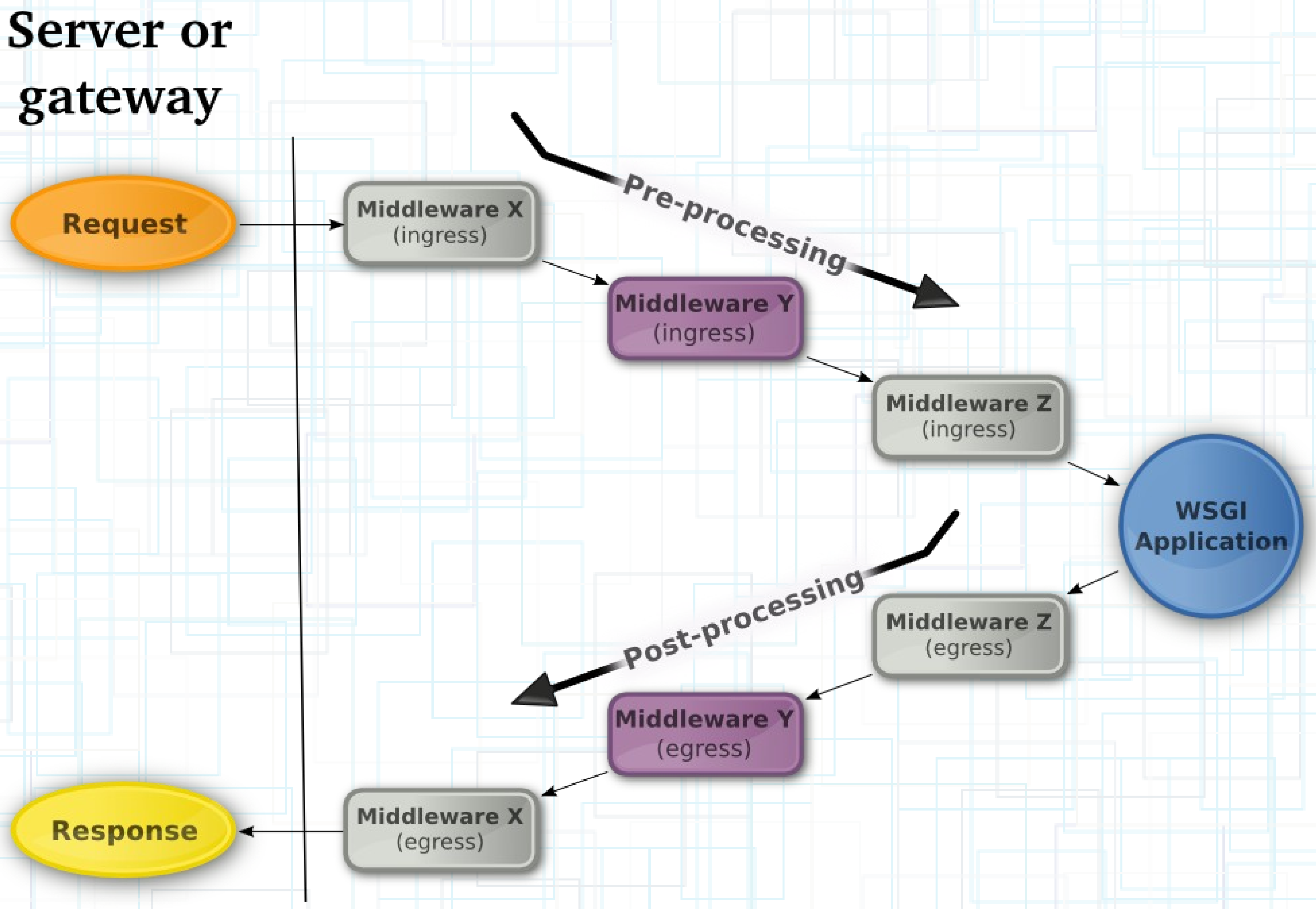
- **Server** with Python embedded:



- Python-powered **gateway**:



Requests and responses



HTTP and WSGI

HTTP requests and responses

Request

```
GET /greeting HTTP/1.1
Host: example.org
User-Agent: EP2010 Client
```

Response

```
HTTP/1.1 200 OK
Server: EP2010 Server
Content-Length: 12
Content-Type: text/plain
```

empty line

```
Hello world!
```

Request

```
POST /login HTTP/1.1
Host: example.org
User-Agent: EP2010 Client
Content-Length: 25
```

empty line

```
username=foo&password=bar
```

Response

```
HTTP/1.1 200 OK
Server: EP2010 Server
Content-Length: 18
Content-Type: text/plain
```

empty line

```
Welcome back, foo!
```

HTTP and WSGI requests

```
POST /login HTTP/1.1
Host: example.org
User-Agent: EP2010 Client
Content-Length: 25
```

empty line

```
username=foo&password=bar
```

```
{
'REQUEST_METHOD': “POST”,
'PATH_INFO': “/login”,
'SERVER_PROTOCOL: “HTTP/1.1”,
'HTTP_HOST': “example.org”,
'HTTP_USER_AGENT': “EP2010 Client”,
'CONTENT_LENGTH': “25”,
'wsgi.input': StringIO(“username=foo&password=bar”),
}
```

WSGI environ variables

They come from:

- CGI (e.g., *PATH_INFO*).
- HTTP request (*HTTP_**).
- WSGI (*wsgi.**).
- Server/gateway (e.g., *mod_wsgi.process_group*).
- 3rd party libraries.
- Yourself.

Raw environ variables

- Request header values are not parsed (some are decoded).
- Some header values are useless as is (e.g., cookies, GET/POST arguments).
- Others are inconvenient as strings (Content Length, If-Modified-Since).
- #1 reason to use a Web framework.

HTTP and WSGI responses

```
HTTP/1.1 200 OK
Server: EP2010 Server
Content-Length: 18
Content-Type: text/plain
```

empty line

```
Welcome back, foo!
```

↓

```
(
  “200 OK”,
  [
    (“Server”, “EP2010 Server”),
    (“Content-Length”, “18”),
    (“Content-Type”, “text/plain”),
  ]
)

[“Welcome back, foo!”]
```

Note that:

- It's not a single object.
- The HTTP version is not set.

WSGI Applications

Simple static application

```
def simple_app(environ, start_response):
    status = "200 OK"
    body = "Hello world!"
    headers = [
        ("Server", "EP2010 Server"),
        ("Content-Length", str(len(body))),
        ("Content-Type", "text/plain"),
    ]

    # Send the headers:
    start_response(status, headers)

    # Now send the body:
    return [body]
```

Response from simple_app()

```
HTTP/1.1 200 OK
Server: EP2010 Server
Content-Length: 12
Content-Type: text/plain
        empty line
Hello world!
```

Simple dynamic application

```
def dynamic_app(environ, start_response):
    headers = [
        ("Content-Type", "text/plain"),
    ]

    if environ['REQUEST_METHOD'] == “GET”:
        status = "200 OK"
        body = "Hello world!"
    else:
        status = "405 Method Not Allowed"
        body = "What are you trying to do?"
        headers.append((“Allow”, “GET”))

    headers.append(("Content-Length", str(len(body))))

    start_response(status, headers)
    return [body]
```

Response from dynamic_app()

```
POST /login HTTP/1.1
Host: example.org
User-Agent: EP2010 Client
Content-Length: 25
```

empty line

```
username=foo&password=bar
```

```
HTTP/1.1 405 Method Not Allowed
Content-Type: text/plain
Allow: GET
Content-Length: 26
```

empty line

```
What are you trying to do?
```

Methods to send the body

- Iterable.
- write() callable; discouraged in new applications.
- File wrapper, to send file-like objects.

Body as an iterable

```
def simple_app(environ, start_response):
    status = "200 OK"
    body = ["Hello", " ", "world", "!"]
    headers = [
        ("Server", "EP2010 Server"),
        ("Content-Length", str(len("".join(body)))),
        ("Content-Type", "text/plain"),
    ]

    # Send the headers:
    start_response(status, headers)

    # Now send the body, without brackets:
    return body
```

The write() callable

```
def simple_app(environ, start_response):
    status = "200 OK"
    body = "Hello world!"
    headers = [
        ("Server", "EP2010 Server"),
        ("Content-Length", str(len(body))),
        ("Content-Type", "text/plain"),
    ]

    # Send the headers and get the writer:
    write = start_response(status, headers)

    # Now send the body:
    write(body)

    # Continue “writing” if necessary...
```

File wrappers

```
FILE = "/tmp/hello.txt"

def simple_app(environ, start_response):
    status = "200 OK"
    fd = open(FILE)
    headers = [
        ("Server", "EP2010 Server"),
        ("Content-Length", str(os.path.getsize(FILE))),
        ("Content-Type", "text/plain"),
    ]

    start_response(status, headers)

    if "wsgi.file_wrapper" in environ:
        return environ['wsgi.file_wrapper'](fd, 1024)
    else:
        return iter(lambda: fd.read(1024), "")
```

WSGI apps in the frameworks

- **CherryPy**: *cherrypy.Application()*
- **Django**: *django.core.handlers.wsgi.WSGIHandler()*
- **Pylons** and **TurboGears 2**: *{PROJECT}.config.middleware.make_app()*
- **Repoze BFG**: *repoze.bfg.paster.get_app()*
- **Zope** 3: *zope.app.wsgi.getWSGIApplication()*

Example

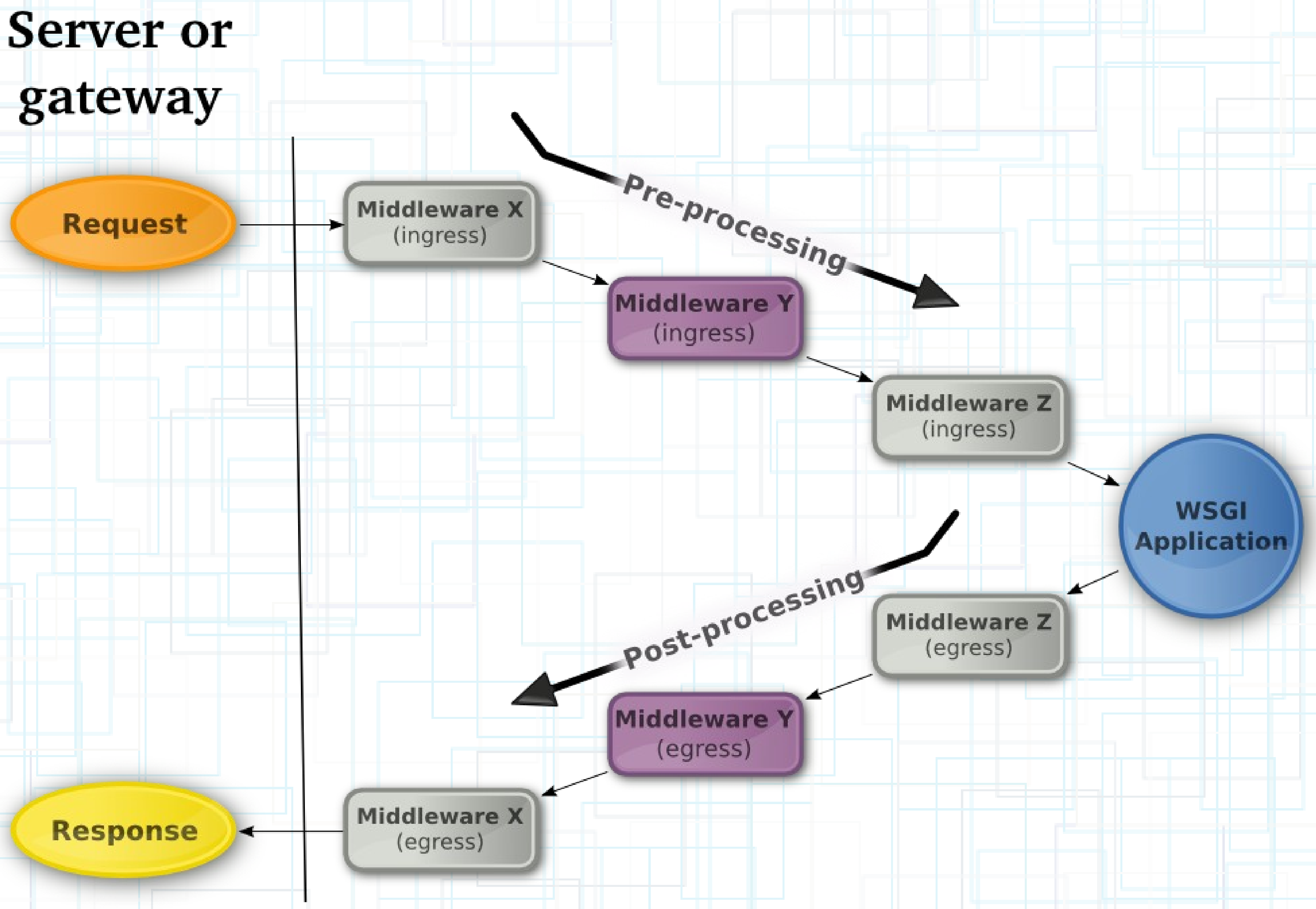
- Open *app_serve_dir.py*.
- See how we return the body with *wsgi.file_wrapper* or just a regular iterable.
- Try it! Run `python app_serve_dir.py'

Interesting/useful applications

- Paste's Proxy, CGI and WaitForIt applications.
- Popular DVCSs: Bazaar and Mercurial.
- Trac, MoinMoin, etc.
- WSGI X-Sendfile.
- twod.wsgi, for Django users.

WSGI Middleware

Do you remember this?



Filtering requests

- Run the WSGI application *conditionally*.
- Change the request the application will receive.
- Add variables to the WSGI environ, which could be consumed by the application later.
- Log them.

Filtering responses

- Add/modify/remove HTTP headers.
- Update the response body.
- Transform the body into something else.
- Log the responses.

Examples

- Open *mw_always_authenticated.py* and try it!
- Then check *mw_wiki_protector.py*.
- See how we can control Trac with WSGI middleware?

Interesting/useful middleware

- Paste's URL mapper, request logger, WDG HTML validator and Lint.
- repoze.who, repoze.what and repoze.profile.
- Routes, Selector or Otto.
- Beaker.
- Many more on pythonpaste.org, repoze.org and wsgi.org.

Testing and debugging

WSGI better than global data

- No global variables. They're evil!
- No messing around with Stdin, Stdout or Stderr. So no *echo* à la PHP!
- The request is just a dictionary.
- The response is made up of a status string, a list of headers and a body iterable.

Functional tests with WebTest

- WebTest is a *functional* test framework for WSGI applications.
- It calls your application directly, without sockets.
- You can inspect the Pythonic response.
- HTML body parsed with BeautifulSoup, ElementTree or lxml.
- Json body parsed with simplejson.
- Try *test_trac.py*.

Debugging techniques

- Inspect the requests and responses (see *mw_debugger.py*).
- Error catching (see *mw_error_catcher.py*).

Handling errors in WSGI

- *environ['wsgi.errors']*: Non-critical errors (see *app_serve_dir_errors.py*).
- *exc_info*: Fatal errors (see *mw_error_catcher.py*).

Embedded Web applications

What can be “embedded”

- Python Web applications.
- Java/PHP/Perl/etc Web applications.
- Standalone web sites.
- Any piece of software with a Web interface.

Why embed applications

- An alternative to many Web server modules.
- Write “middleware” for them.
- For example, Single Sign-On, authorization.

Example embedded applications

- Try the Single Sign-On system between Django and Trac in the *Weesgo* application.
- Try running the PHP-powered WordPress under WSGI (*run_wordpress.py*).

Deployment

Server advantages

- Embedded.
- Usually easier to set up.
- Better performance.

Gateway advantages

- Non-embedded.
- Applications can be run by different users.
- No need to restart the Web server to upgrade code.
- Applications with different versions of Python.
- No shared libraries conflicts.

Server examples

- Apache + mod_wsgi
- Gunicorn.
- Tornado.
- Paste Script (paster).
- Django's `manage runserver'

Gateway examples

- CGI and the like (FastCGI, SCGI).
- Apache JServ Protocol (AJP).
- Apache + mod_wsgi in daemon mode (one Python version limitation still present).

The fine print (Limitations)

No Python 3 support

- Bytes vs (unicode) strings.
- Bytes don't behave like strings anymore.
- WSGI 1.0 is based on bytes (*str* in Python 2).
- No consensus, yet. But getting there.

Decoded values

- CGI requires paths to be decoded (those %XX strings in the URL).
- Cannot distinguish %2F from /
- Browsers don't help either.

No unknown length wsgi.input

- Some libraries use CONTENT_LENGTH=-1
- Others use “0”, which actually means “there are no bytes in wsgi.input”.
- The right way to do it is in a chucked request content, with “Transfer-Encoding: chunked”. But it's not part of WSGI 1.0.

Conclusion

Summary

- WSGI means interoperability.
- More software for you to use.
- Pretty much everybody uses WSGI; even unconsciously.
- WSGI 1.0 is not perfect.
- We've basically covered PEP-333.

What I didn't talk about

- wsgiref: Like the Paste project, with less functionality. But it's part of the stdlib.
- Details in PEP-333 which I didn't find interesting for application developers.
- mod_python: It's not WSGI and it's dead.

Frameworks are not the only true answer

- Thanks to WSGI:
  - Pythonic wrappers for the requests and responses: WebOb.
  - Request dispatchers: Routes, Selector, Otto.
  - Auth: repoze.who and repoze.what.
  - Sessions: Beaker.
- WSGI-independent:
  - ORM: SQLAlchemy, Elixir, SQLObject.
  - Templates: Jinja, Mako, etc.
  - Form validation: FormEncode.

That's it. Thanks!