wptserve Documentation

Release 0.1

James Graham

Jun 10, 2017

Contents

A python-based HTTP server specifically targeted at being used for testing the web platform. This means that extreme flexibility — including the possibility of HTTP non-conformance — in the response is supported.

Contents:

CHAPTER 1

Introduction

wptserve has been designed with the specific goal of making a server that is suitable for writing tests for the web platform. This means that it cannot use common abstractions over HTTP such as WSGI, since these assume that the goal is to generate a well-formed HTTP response. Testcases, however, often require precise control of the exact bytes sent over the wire and their timing. The full list of design goals for the server are:

- Suitable to run on individual test machines and over the public internet.
- Support plain TCP and SSL servers.
- Serve static files with the minimum of configuration.
- Allow headers to be overwritten on a per-file and per-directory basis.
- Full customisation of headers sent (e.g. altering or omitting "mandatory" headers).
- Simple per-client state.
- Complex logic in tests, up to precise control over the individual bytes sent and the timing of sending them.

Request Handling

At the high level, the design of the server is based around similar concepts to those found in common web frameworks like Django, Pyramid or Flask. In particular the lifecycle of a typical request will be familiar to users of these systems. Incoming requests are parsed and a *Request* object is constructed. This object is passed to a *Router* instance, which is responsible for mapping the request method and path to a handler function. This handler is passed two arguments; the request object and a *Response* object. In cases where only simple responses are required, the handler function may fill in the properties of the response object and the server will take care of constructing the response. However each Response also contains a *ResponseWriter* which can be used to directly control the TCP socket.

By default there are several built-in handler functions that provide a higher level API than direct manipulation of the Response object. These are documented in *Handlers*.

CHAPTER 2

Server

Basic server classes and router.

The following example creates a server that serves static files from the *files* subdirectory of the current directory and causes it to run on port 8080 until it is killed:

```
from wptserve import server, handlers

httpd = server.WebTestHttpd(port=8080, doc_root="./files/",
                            routes=[("GET", "*", handlers.file_handler)])
httpd.start(block=True)
```

Interface

class wptserve.server.**WebTestHttpd**(*host='127.0.0.1'*, *port=8000*, *server_cls=None*, *handler_cls=<class wptserve.server.WebTestRequestHandler>*, *use_ssl=False*, *key_file=None*, *certificate=None*, *encrypt_after_connect=False*, *router_cls=<class 'wptserve.router.Router'>*, *doc_root='.'*, *routes=None*, *rewriter_cls=<class 'wptserve.server.RequestRewriter'>*, *bind_hostname=True*, *rewrites=None*, *latency=None*, *config=None*)

Parameters

- **host** – Host from which to serve (default: 127.0.0.1)
- **port** – Port from which to serve (default: 8000)
- **server_cls** – Class to use for the server (default depends on ssl vs non-ssl)
- **handler_cls** – Class to use for the RequestHandler
- **use_ssl** – Use a SSL server if no explicit server_cls is supplied
- **key_file** – Path to key file to use if ssl is enabled

- **certificate** – Path to certificate file to use if ssl is enabled
- **encrypt_after_connect** – For each connection, don't start encryption until a CONNECT message has been received. This enables the server to act as a self-proxy.
- **router_cls** – Router class to use when matching URLs to handlers
- **doc_root** – Document root for serving files
- **routes** – List of routes with which to initialize the router
- **rewriter_cls** – Class to use for request rewriter
- **rewrites** – List of rewrites with which to initialize the rewriter_cls
- **config** – Dictionary holding environment configuration settings for handlers to read, or None to use the default values.
- **bind_hostname** – Boolean indicating whether to bind server to hostname.
- **latency** – Delay in ms to wait before seving each response, or callable that returns a delay in ms

HTTP server designed for testing scenarios.

Takes a router class which provides one method get_handler which takes a Request and returns a handler function.

host
The host name or ip address of the server

port
The port on which the server is running

router
The Router object used to associate requests with resources for this server

rewriter
The Rewriter object used for URL rewriting

use_ssl
Boolean indicating whether the server is using ssl

started
Boolean indictaing whether the server is running

start(*block=False*)
Start the server.

Parameters **block** – True to run the server on the current thread, blocking, False to run on a separate thread.

stop()
Stops the server.

If the server is not running, this method has no effect.

class wptserve.server.**WebTestRequestHandler**(*request*, *client_address*, *server*)
RequestHandler for WebTestHttpd

CHAPTER 3

Router

The router is used to match incoming requests to request handler functions. Typically users don't interact with the router directly, but instead send a list of routes to register when starting the server. However it is also possible to add routes after starting the server by calling the *register* method on the server's *router* property.

Routes are represented by a three item tuple:

```
(methods, path_match, handler)
```

methods is either a string or a list of strings indicating the HTTP methods to match. In cases where all methods should match there is a special sentinel value *any_method* provided as a property of the *router* module that can be used.

path_match is an expression that will be evaluated against the request path to decide if the handler should match. These expressions follow a custom syntax intended to make matching URLs straightforward and, in particular, to be easier to use than raw regexp for URL matching. There are three possible components of a match expression:

- Literals. These match any character. The special characters *, { and } must be escaped by prefixing them with a \.
- Match groups. These match any character other than / and save the result as a named group. They are delimited by curly braces; for example:

  ```
  {abc}
  ```

  would create a match group with the name *abc*.
- Stars. These are denoted with a * and match any character including /. There can be at most one star per pattern and it must follow any match groups.

Path expressions always match the entire request path and a leading / in the expression is implied even if it is not explicitly provided. This means that */foo* and *foo* are equivalent.

For example, the following pattern matches all requests for resources with the extension *.py*:

```
*.py
```

The following expression matches anything directly under */resources* with a *.html* extension, and places the "filename" in the *name* group:

```
/resources/{name}.html
```

The groups, including anything that matches a * are available in the request object through the *route_match* property. This is a dictionary mapping the group names, and any match for * to the matching part of the route. For example, given a route:

```
/api/{sub_api}/*
```

and the request path */api/test/html/test.html*, *route_match* would be:

```
{"sub_api": "html", "*": "html/test.html"}
```

handler is a function taking a request and a response object that is responsible for constructing the response to the HTTP request. See *Handlers* for more details on handler functions.

Interface

class wptserve.router.**Router**(*doc_root*, *routes*)
Object for matching handler functions to requests.

Parameters

- **doc_root** – Absolute path of the filesystem location from which to serve tests
- **routes** – Initial routes to add; a list of three item tuples (method, path_pattern, handler_function), defined as for register()

get_handler(*request*)
Get a handler for a request or None if there is no handler.

Parameters **request** – Request to get a handler for.

Return type Callable or None

register(*methods*, *path*, *handler*)
Register a handler for a set of paths.

Parameters

- **methods** – Set of methods this should match. "*" is a special value indicating that all methods should be matched.
- **path_pattern** –

Match pattern that will be used to determine if a request path matches this route. Match patterns consist of either literal text, match groups, denoted {name}, which match any character except /, and, at most one *, which matches and character and creates a match group to the end of the string. If there is no leading "/" on the pattern, this is automatically implied. For example:

```
api/{resource}/*.json
```

Would match */api/test/data.json* or */api/test/test2/data.json*, but not */api/test/data.py*.

The match groups are made available in the request object as a dictionary through the route_match property. For example, given the route pattern above and the path */api/test/data.json*, the route_match property would contain:

```
{"resource": "test", "*": "data.json"}
```

- **handler** – Function that will be called to process matching requests. This must take two parameters, the request object and the response object.

wptserve.router.**compile_path_match**(*route_pattern*)

tokens: / or literal or match or *

CHAPTER 4

Request

Request object.

Interface

class wptserve.request.**Authentication**(*headers*)
Object for dealing with HTTP Authentication

username

The username supplied in the HTTP Authorization header, or None

password

The password supplied in the HTTP Authorization header, or None

class wptserve.request.**CookieValue**(*morsel*)
Representation of cookies.

Note that cookies are considered read-only and the string value of the cookie will not change if you update the field values. However this is not enforced.

key

The name of the cookie.

value

The value of the cookie

expires

The expiry date of the cookie

path

The path of the cookie

comment

The comment of the cookie.

domain

The domain with which the cookie is associated

max_age

The max-age value of the cookie.

secure

Whether the cookie is marked as secure

httponly

Whether the cookie is marked as httponly

class wptserve.request.**Cookies**
MultiDict specialised for Cookie values

class wptserve.request.**MultiDict**
Dictionary type that holds multiple values for each key

first(*key*, *default=<object object>*)
Get the first value with a given key

Parameters

- **key** – The key to lookup
- **default** – The default to return if key is not found (throws if nothing is specified)

get_list(*key*)
Get all values with a given key as a list

Parameters **key** – The key to lookup

last(*key*, *default=<object object>*)
Get the last value with a given key

Parameters

- **key** – The key to lookup
- **default** – The default to return if key is not found (throws if nothing is specified)

class wptserve.request.**Request**(*request_handler*)
Object representing a HTTP request.

doc_root

The local directory to use as a base when resolving paths

route_match

Regexp match object from matching the request path to the route selected for the request.

protocol_version

HTTP version specified in the request.

method

HTTP method in the request.

request_path

Request path as it appears in the HTTP request.

url_base

The prefix part of the path; typically / unless the handler has a url_base set

url

Absolute URL for the request.

url_parts

Parts of the requested URL as obtained by urlparse.urlsplit(path)

request_line

Raw request line

headers

RequestHeaders object providing a dictionary-like representation of the request headers.

raw_headers.

Dictionary of non-normalized request headers.

body

Request body as a string

raw_input

File-like object representing the body of the request.

GET

MultiDict representing the parameters supplied with the request. Note that these may be present on non-GET requests; the name is chosen to be familiar to users of other systems such as PHP.

POST

MultiDict representing the request body parameters. Most parameters are present as string values, but file uploads have file-like values.

cookies

Cookies object representing cookies sent with the request with a dictionary-like interface.

auth

Object with username and password properties representing any credentials supplied using HTTP authentication.

server

Server object containing information about the server environment.

class wptserve.request.**RequestHeaders**(*items*)
Dictionary-like API for accessing request headers.

get(*key*, *default=None*)
Get a string representing all headers with a particular value, with multiple headers separated by a comma. If no header is found return a default value

Parameters

- **key** – The header name to look up (case-insensitive)
- **default** – The value to return in the case of no match

get_list(*key*, *default=<object object>*)
Get all the header values for a particular field name as a list

class wptserve.request.**Server**(*request*)
Data about the server environment

config

Environment configuration information with information about the various servers running, their hostnames and ports.

stash

Stash object holding state stored on the server between requests.

CHAPTER 5

Response

Response object. This object is used to control the response that will be sent to the HTTP client. A handler function will take the response object and fill in various parts of the response. For example, a plain text response with the body 'Some example content' could be produced as:

```
def handler(request, response):
    response.headers.set("Content-Type", "text/plain")
    response.content = "Some example content"
```

The response object also gives access to a ResponseWriter, which allows direct access to the response socket. For example, one could write a similar response but with more explicit control as follows:

```
import time

def handler(request, response):
    response.add_required_headers = False # Don't implicitly add HTTP headers
    response.writer.write_status(200)
    response.writer.write_header("Content-Type", "text/plain")
    response.writer.write_header("Content-Length", len("Some example content"))
    response.writer.end_headers()
    response.writer.write("Some ")
    time.sleep(1)
    response.writer.write("example content")
```

Note that when writing the response directly like this it is always necessary to either set the Content-Length header or set *response.close_connection = True*. Without one of these, the client will not be able to determine where the response body ends and will continue to load indefinitely.

Interface

class wptserve.response.**Response**(*handler*, *request*)
Object representing the response to a HTTP request

Parameters

- **handler** – RequestHandler being used for this response
- **request** – Request that this is the response for

request
Request associated with this Response.

encoding
The encoding to use when converting unicode to strings for output.

add_required_headers
Boolean indicating whether mandatory headers should be added to the response.

send_body_for_head_request
Boolean, default False, indicating whether the body content should be sent when the request method is HEAD.

explicit_flush
Boolean indicating whether output should be flushed automatically or only when requested.

writer
The ResponseWriter for this response

status
Status tuple (code, message). Can be set to an integer, in which case the message part is filled in automatically, or a tuple.

headers
List of HTTP headers to send with the response. Each item in the list is a tuple of (name, value).

content
The body of the response. This can either be a string or a iterable of response parts. If it is an iterable, any item may be a string or a function of zero parameters which, when called, returns a string.

delete_cookie(*name*, *path='/'*, *domain=None*)
Delete a cookie on the client by setting it to the empty string and to expire in the past

iter_content(*read_file=False*)
Iterator returning chunks of response body content.

If any part of the content is a function, this will be called and the resulting value (if any) returned.

Parameters **read_file** –

- boolean controlling the behaviour when content

is a file handle. When set to False the handle will be returned directly allowing the file to be passed to the output in small chunks. When set to True, the entire content of the file will be returned as a string facilitating non-streaming operations like template substitution.

set_cookie(*name*, *value*, *path='/'*, *domain=None*, *max_age=None*, *expires=None*, *secure=False*, *httponly=False*, *comment=None*)
Set a cookie to be sent with a Set-Cookie header in the response

Parameters

- **name** – String name of the cookie
- **value** – String value of the cookie
- **max_age** – datetime.timedelta int representing the time (in seconds) until the cookie expires
- **path** – String path to which the cookie applies
- **domain** – String domain to which the cookie applies

- **secure** – Boolean indicating whether the cookie is marked as secure
- **httponly** – Boolean indicating whether the cookie is marked as HTTP Only
- **comment** – String comment
- **expires** – datetime.datetime or datetime.timedelta indicating a time or interval from now when the cookie expires

set_error(*code*, *message=''*)
Set the response status headers and body to indicate an error

unset_cookie(*name*)
Remove a cookie from those that are being sent with the response

write()
Write the whole response

write_content()
Write out the response content

write_status_headers()
Write out the status line and headers for the response

class wptserve.response.**ResponseHeaders**
Dictionary-like object holding the headers for the response

append(*key*, *value*)
Add a new header with a given name, not overwriting any existing headers with the same name

Parameters

- **key** – Name of the header to add
- **value** – Value to set for the header

get(*key*, *default=<object object>*)
Get the set values for a particular header.

set(*key*, *value*)
Set a header to a specific value, overwriting any previous header with the same name

Parameters

- **key** – Name of the header to set
- **value** – Value to set the header to

class wptserve.response.**ResponseWriter**(*handler*, *response*)
Object providing an API to write out a HTTP response.

Parameters

- **handler** – The RequestHandler being used.
- **response** – The Response associated with this writer.

After each part of the response is written, the output is flushed unless response.explicit_flush is False, in which case the user must call .flush() explicitly.

encode(*data*)
Convert unicode to bytes according to response.encoding.

end_headers()
Finish writing headers and write the separator.

Unless add_required_headers on the response is False, this will also add HTTP-mandated headers that have not yet been supplied to the response headers

flush()
Flush the output.

write(*data*)
Write directly to the response, converting unicode to bytes according to response.encoding. Does not flush.

write_content(*data*)
Write the body of the response.

write_content_file(*data*)
Write a file-like object directly to the response in chunks. Does not flush.

write_header(*name*, *value*)
Write out a single header for the response.

Parameters

- **name** – Name of the header field
- **value** – Value of the header field

write_status(*code*, *message=None*)
Write out the status line of a response.

Parameters

- **code** – The integer status code of the response.
- **message** – The message of the response. Defaults to the message commonly used with the status code.

CHAPTER 6

Stash

Object for storing cross-request state. This is unusual in that keys must be UUIDs, in order to prevent different clients setting the same key, and values are write-once, read-once to minimise the chances of state persisting indefinitely. The stash defines two operations; *put*, to add state and *take* to remove state. Furthermore, the view of the stash is path-specific; by default a request will only see the part of the stash corresponding to its own path.

A typical example of using a stash to store state might be:

```
@handler
def handler(request, response):
    # We assume this is a string representing a UUID
    key = request.GET.first("id")

    if request.method == "POST":
        request.server.stash.put(key, "Some sample value")
        return "Added value to stash"
    else:
        value = request.server.stash.take(key)
        assert request.server.stash.take(key) is None
        return key
```

Interface

class wptserve.stash.**Stash**(*default_path*, *address=None*, *authkey=None*)

Key-value store for persisting data across HTTP/S and WS/S requests.

This data store is specifically designed for persisting data across server requests. The synchronization is achieved by using the BaseManager from the multiprocessing module so different processes can acccess the same data.

Stash can be used interchangeably between HTTP, HTTPS, WS and WSS servers. A thing to note about WS/S servers is that they require additional steps in the handlers for accessing the same underlying shared data in the Stash. This can usually be achieved by using load_env_config(). When using Stash interchangeably between HTTP/S and WS/S request, the path part of the key should be expliclitly specified if accessing the same key/value subset.

The store has several unusual properties. Keys are of the form (path, uuid), where path is, by default, the path in the HTTP request and uuid is a unique id. In addition, the store is write-once, read-once, i.e. the value associated with a particular key cannot be changed once written and the read operation (called "take") is destructive. Taken together, these properties make it difficult for data to accidentally leak between different resources or different requests for the same resource.

put(*key*, *value*, *path=None*)
Place a value in the shared stash.

Parameters

- **key** – A UUID to use as the data's key.
- **value** – The data to store. This can be any python object.
- **path** – The path that has access to read the data (by default the current request path)

take(*key*, *path=None*)
Remove a value from the shared stash and return it.

Parameters

- **key** – A UUID to use as the data's key.
- **path** – The path that has access to read the data (by default the current request path)

CHAPTER 7

Handlers

Handlers are functions that have the general signature:

```
handler(request, response)
```

It is expected that the handler will use information from the request (e.g. the path) either to populate the response object with the data to send, or to directly write to the output stream via the ResponseWriter instance associated with the request. If a handler writes to the output stream then the server will not attempt additional writes, i.e. the choice to write directly in the handler or not is all-or-nothing.

A number of general-purpose handler functions are provided by default:

Python Handlers

Python handlers are functions which provide a higher-level API over manually updating the response object, by causing the return value of the function to provide (part of) the response. There are three possible sets of values that may be returned:

```
(status, headers, content)
(headers, content)
content
```

Here *status* is either a tuple (status code, message) or simply a integer status code, *headers* is a list of (field name, value) pairs, and *content* is a string or an iterable returning strings. Such a function may also update the response manually. For example one may use *response.headers.set* to set a response header, and only return the content. One may even use this kind of handler, but manipulate the output socket directly, in which case the return value of the function, and the properties of the response object, will be ignored.

The most common way to make a user function into a python handler is to use the provided *wptserve.handlers.handler* decorator:

```
from wptserve.handlers import handler
```

```
@handler
def test(request, response):
    return [("X-Test": "PASS"), ("Content-Type", "text/plain")], "test"

#Later, assuming we have a Router object called 'router'

router.register("GET", "/test", test)
```

JSON Handlers

This is a specialisation of the python handler type specifically designed to facilitate providing JSON responses. The API is largely the same as for a normal python handler, but the *content* part of the return value is JSON encoded, and a default Content-Type header of *application/json* is added. Again this handler is usually used as a decorator:

```
from wptserve.handlers import json_handler

@json_handler
def test(request, response):
    return {"test": "PASS"}
```

Python File Handlers

Python file handlers are designed to provide a vaguely PHP-like interface where each resource corresponds to a particular python file on the filesystem. Typically this is hooked up to a route like `("*", "*.py", python_file_handler)`, meaning that any .py file will be treated as a handler file (note that this makes python files unsafe in much the same way that .php files are when using PHP).

Unlike PHP, the python files don't work by outputting text to stdout from the global scope. Instead they must define a single function *main* with the signature:

```
main(request, response)
```

This function then behaves just like those described in *Python Handlers* above.

asis Handlers

These are used to serve files as literal byte streams including the HTTP status line, headers and body. In the default configuration this handler is invoked for all files with a .asis extension.

File Handlers

File handlers are used to serve static files. By default the content type of these files is set by examining the file extension. However this can be overridden, or additional headers supplied, by providing a file with the same name as the file being served but an additional .headers suffix, i.e. test.html has its headers set from test.html.headers. The format of the .headers file is plaintext, with each line containing:

```
Header-Name: header_value
```

In addition headers can be set for a whole directory of files (but not subdirectories), using a file called *__dir__.headers*.

CHAPTER 8

Pipes

Pipes are functions that may be used when serving files to alter parts of the response. These are invoked by adding a pipe= query parameter taking a | separated list of pipe functions and parameters. The pipe functions are applied to the response from left to right. For example:

```
GET /sample.txt?pipe=slice(1,200)|status(404).
```

This would serve bytes 1 to 199, inclusive, of foo.txt with the HTTP status code 404.

There are several built-in pipe functions, and it is possible to add more using the *@pipe* decorator on a function, if required.

Note: Because of the way pipes compose, using some pipe functions prevents the content-length of the response from being known in advance. In these cases the server will close the connection to indicate the end of the response, preventing the use of HTTP 1.1 keepalive.

Built-In Pipes

sub

Used to substitute variables from the server environment, or from the request into the response.

Substitutions are marked in a file using a block delimited by *{{* and *}}*. Inside the block the following variables are available:

{{host}} The host name of the server excluding any subdomain part.

{{domains[]}}

The domain name of a particular subdomain e.g. *{{domains[www]}}* for the *www* subdomain.

{{ports[][]}}

The port number of servers, by protocol e.g. *{{ports[http][0]}}* for the first (and, depending on setup, possibly only) http server

{{headers[]}}

The HTTP headers in the request e.g. *{{headers[X-Test]}}* for a hypothetical *X-Test* header.

{{GET[]}}

The query parameters for the request e.g. *{{GET[id]}}* for an id parameter sent with the request.

So, for example, to write a javascript file called *xhr.js* that depends on the host name of the server, without hardcoding, one might write:

```
var server_url = http://{{host}}:{{ports[http][0]}}/path/to/resource;
//Create the actual XHR and so on
```

The file would then be included as:

<script src="xhr.js?pipe=sub"></script>

This pipe can also be enabled by using a filename **.sub.ext*, e.g. the file above could be called *xhr.sub.js*.

status

Used to set the HTTP status of the response, for example:

```
example.js?pipe=status(410)
```

headers

Used to add or replace http headers in the response. Takes two or three arguments; the header name, the header value and whether to append the header rather than replace an existing header (default: False). So, for example, a request for:

```
example.html?pipe=header(Content-Type,text/plain)
```

causes example.html to be returned with a text/plain content type whereas:

```
example.html?pipe=header(Content-Type,text/plain,True)
```

Will cause example.html to be returned with both text/html and text/plain content-type headers.

slice

Used to send only part of a response body. Takes the start and, optionally, end bytes as arguments, although either can be null to indicate the start or end of the file, respectively. So for example:

```
example.txt?pipe=slice(10,20)
```

Would result in a response with a body containing 10 bytes of example.txt including byte 10 but excluding byte 20.

```
example.txt?pipe=slice(10)
```

Would cause all bytes from byte 10 of example.txt to be sent, but:

```
example.txt?pipe=slice(null,20)
```

Would send the first 20 bytes of example.txt.

trickle

Note: Using this function will force a connection close.

Used to send the body of a response in chunks with delays. Takes a single argument that is a microsyntax consisting of colon-separated commands. There are three types of commands:

- Bare numbers represent a number of bytes to send
- Numbers prefixed *d* indicate a delay in seconds
- Numbers prefixed *r* must only appear at the end of the command, and indicate that the preceding N items must be repeated until there is no more content to send. The number of items to repeat must be even.

In the absence of a repetition command, the entire remainder of the content is sent at once when the command list is exhausted. So for example:

```
example.txt?pipe=trickle(d1)
```

causes a 1s delay before sending the entirety of example.txt.

```
example.txt?pipe=trickle(100:d1)
```

causes 100 bytes of example.txt to be sent, followed by a 1s delay, and then the remainder of the file to be sent. On the other hand:

```
example.txt?pipe=trickle(100:d1:r2)
```

Will cause the file to be sent in 100 byte chunks separated by a 1s delay until the whole content has been sent.

Interface

wptserve.pipes.**gzip**(*request*, *response*)
This pipe gzip-encodes response data.

It sets (or overwrites) these HTTP headers: Content-Encoding is set to gzip Content-Length is set to the length of the compressed content

wptserve.pipes.**header**(*request*, *response*, *name*, *value*, *append=False*)
Set a HTTP header.

Replaces any existing HTTP header of the same name unless append is set, in which case the header is appended without replacement.

Parameters

- **name** – Name of the header to set.
- **value** – Value to use for the header.
- **append** – True if existing headers should not be replaced

wptserve.pipes.**slice**(*request*, *response*, *start*, *end=None*)
Send a byte range of the response body

Parameters

- **start** – The starting offset. Follows python semantics including negative numbers.
- **end** – The ending offset, again with python semantics and None (spelled "null" in a query string) to indicate the end of the file.

wptserve.pipes.**status**(*request*, *response*, *code*)
Alter the status code.

Parameters **code** – Status code to use for the response.

wptserve.pipes.**sub**(*request*, *response*, *escape_type='html'*)
Substitute environment information about the server and request into the script.

Parameters **escape_type** – String detailing the type of escaping to use. Known values are "html" and "none", with "html" the default for historic reasons.

The format is a very limited template language. Substitutions are enclosed by {{ and }}. There are several avaliable substitutions:

host A simple string value and represents the primary host from which the tests are being run.

domains A dictionary of available domains indexed by subdomain name.

ports A dictionary of lists of ports indexed by protocol.

location A dictionary of parts of the request URL. Valid keys are 'server, 'scheme', 'host', 'hostname', 'port', 'path' and 'query'. 'server' is scheme://host:port, 'host' is hostname:port, and query

includes the leading '?', but other delimiters are omitted.

headers A dictionary of HTTP headers in the request.

GET A dictionary of query parameters supplied with the request.

uuid() A pesudo-random UUID suitable for usage with stash

So for example in a setup running on localhost with a www subdomain and a http server on ports 80 and 81:

```
{{host}} => localhost
{{domains[www]}} => www.localhost
{{ports[http][1]}} => 81
```

It is also possible to assign a value to a variable name, which must start with the $ character, using the ":" syntax e.g.

{{$id:uuid()}

Later substitutions in the same file may then refer to the variable by name e.g.

{{$id}}

wptserve.pipes.**trickle**(*request*, *response*, *delays*)
Send the response in parts, with time delays.

Parameters **delays** – A string of delays and amounts, in bytes, of the response to send. Each component is separated by a colon. Amounts in bytes are plain integers, whilst delays are floats prefixed with a single d e.g. d1:100:d2 Would cause a 1 second delay, would then send 100 bytes of the file, and then cause a 2 second delay, before sending the remainder of the file.

If the last token is of the form rN, instead of sending the remainder of the file, the previous N instructions will be repeated until the whole file has been sent e.g. d1:100:d2:r2 Causes a delay

of 1s, then 100 bytes to be sent, then a 2s delay and then a further 100 bytes followed by a two second delay until the response has been fully sent.

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Python Module Index

w

Index

R

S

T

U

V

W