The WSGI Reference Library

Release 0.2

Phillip J. Eby

October 4, 2010

Email: pje@telecommunity.com

Abstract

The Web Server Gateway Interface (WSGI) is a standard interface between web server software and web applications written in Python. Having a standard interface makes it easy to use an application that supports WSGI with a number of different web servers.

`wsgiref` is a reference implementation of the WSGI specification that can be used to add WSGI support to a web server or framework. It also contains some useful utilities for writing applications or middleware that provide or implement WSGI.

CONTENTS

CHAPTER

ONE

Reference

1.1 `wsgiref` — WSGI Utilities and Reference Implementation

The Web Server Gateway Interface (WSGI) is a standard interface between web server software and web applications written in Python. Having a standard interface makes it easy to use an application that supports WSGI with a number of different web servers.

Only authors of web servers and programming frameworks need to know every detail and corner case of the WSGI design. You don't need to understand every detail of WSGI just to install a WSGI application or to write a web application using an existing framework.

`wsgiref` is a reference implementation of the WSGI specification that can be used to add WSGI support to a web server or framework. It provides utilities for manipulating WSGI environment variables and response headers, base classes for implementing WSGI servers, a demo HTTP server that serves WSGI applications, and a validation tool that checks WSGI servers and applications for conformance to the WSGI specification (PEP 333).

1.1.1 `wsgiref.util` – WSGI environment utilities

This module provides a variety of utility functions for working with WSGI environments. A WSGI environment is a dictionary containing HTTP request variables as described in PEP 333. All of the functions taking an *environ* parameter expect a WSGI-compliant dictionary to be supplied; please see PEP 333 for a detailed specification.

`guess_scheme`(*environ*)
: Return a guess for whether `wsgi.url_scheme` should be "http" or "https", by checking for a `HTTPS` environment variable in the *environ* dictionary. The return value is a string.

  This function is useful when creating a gateway that wraps CGI or a CGI-like protocol such as FastCGI. Typically, servers providing such protocols will include a `HTTPS` variable with a value of "1" "yes", or "on" when a request is received via SSL. So, this function returns "https" if such a value is found, and "http" otherwise.

`request_uri`(*environ* [, *include_query=1*])
: Return the full request URI, optionally including the query string, using the algorithm found in the "URL Reconstruction" section of PEP 333. If *include_query* is false, the query string is not included in the resulting URI.

`application_uri`(*environ*)
: Similar to `request_uri`, except that the `PATH_INFO` and `QUERY_STRING` variables are ignored. The result is the base URI of the application object addressed by the request.

`shift_path_info`(*environ*)
: Shift a single name from `PATH_INFO` to `SCRIPT_NAME` and return the name. The *environ* dictionary is *modified* in-place; use a copy if you need to keep the original `PATH_INFO` or `SCRIPT_NAME` intact.

  If there are no remaining path segments in `PATH_INFO`, `None` is returned.

Typically, this routine is used to process each portion of a request URI path, for example to treat the path as a series of dictionary keys. This routine modifies the passed-in environment to make it suitable for invoking another WSGI application that is located at the target URI. For example, if there is a WSGI application at `/foo`, and the request URI path is `/foo/bar/baz`, and the WSGI application at `/foo` calls `shift_path_info`, it will receive the string "bar", and the environment will be updated to be suitable for passing to a WSGI application at `/foo/bar`. That is, `SCRIPT_NAME` will change from `/foo` to `/foo/bar`, and `PATH_INFO` will change from `/bar/baz` to `/baz`.

When `PATH_INFO` is just a "/", this routine returns an empty string and appends a trailing slash to `SCRIPT_NAME`, even though empty path segments are normally ignored, and `SCRIPT_NAME` doesn't normally end in a slash. This is intentional behavior, to ensure that an application can tell the difference between URIs ending in `/x` from ones ending in `/x/` when using this routine to do object traversal.

`setup_testing_defaults`(*environ*)

Update *environ* with trivial defaults for testing purposes.

This routine adds various parameters required for WSGI, including `HTTP_HOST`, `SERVER_NAME`, `SERVER_PORT`, `REQUEST_METHOD`, `SCRIPT_NAME`, `PATH_INFO`, and all of the PEP 333-defined `wsgi.*` variables. It only supplies default values, and does not replace any existing settings for these variables.

This routine is intended to make it easier for unit tests of WSGI servers and applications to set up dummy environments. It should NOT be used by actual WSGI servers or applications, since the data is fake!

In addition to the environment functions above, the `wsgiref.util` module also provides these miscellaneous utilities:

`is_hop_by_hop`(*header_name*)

Return true if 'header_name' is an HTTP/1.1 "Hop-by-Hop" header, as defined by RFC 2616.

class `FileWrapper`(*filelike* [, *blksize=8192*])

A wrapper to convert a file-like object to an iterator. The resulting objects support both `__getitem__` and `__iter__` iteration styles, for compatibility with Python 2.1 and Jython. As the object is iterated over, the optional *blksize* parameter will be repeatedly passed to the *filelike* object's `read()` method to obtain strings to yield. When `read()` returns an empty string, iteration is ended and is not resumable.

If *filelike* has a `close()` method, the returned object will also have a `close()` method, and it will invoke the *filelike* object's `close()` method when called.

`test`(*app, environ={}, form={}, **kw*)

Print the output of a WSGI app (e.g. for use in doctests)

Runs *app* as a WSGI application and prints its output. If an untrapped error occurs in *app*, it drops into the `pdb` debugger's post-mortem debug shell (using `sys.__stdout__` if `sys.stdout` has been replaced).

Any keyword arguments are added to the environment used to run *app*. If a keyword argument begins with `wsgi_`, the `_` is replaced with a `.`, so that you can set e.g. `wsgi.multithread` using a `wsgi_multithread` keyword argument.

If a non-empty *form* dictionary is provided, it is treated as a collection of fields for a form `POST`. The `REQUEST_METHOD` will default to `POST`, and the default `CONTENT_LENGTH`, `CONTENT_TYPE`, and `wsgi.input` values will be appropriately set (but can still be overridden by explicit keyword arguments or the *environ* argument).

Any *form* values that are not instances of `basestring` are assumed to be *sequences* of values, and will result in multiple name/value pairs being added to the encoded data sent to the application.

Any WSGI-required variables that are not specified by *environ*, *form*, or keyword arguments, are initialized to default values using the `setup_testing_defaults()` function.

(New in version 0.2.)

1.1.2 wsgiref.headers – WSGI response header tools

This module provides a single class, Headers, for convenient manipulation of WSGI response headers using a mapping-like interface.

class Headers(*headers*)

Create a mapping-like object wrapping *headers*, which must be a list of header name/value tuples as described in PEP 333. Any changes made to the new Headers object will directly update the *headers* list it was created with.

Headers objects support typical mapping operations including __getitem__, get, __setitem__, setdefault, __delitem__, __contains__ and has_key. For each of these methods, the key is the header name (treated case-insensitively), and the value is the first value associated with that header name. Setting a header deletes any existing values for that header, then adds a new value at the end of the wrapped header list. Headers' existing order is generally maintained, with new headers added to the end of the wrapped list.

Unlike a dictionary, Headers objects do not raise an error when you try to get or delete a key that isn't in the wrapped header list. Getting a nonexistent header just returns None, and deleting a nonexistent header does nothing.

Headers objects also support keys(), values(), and items() methods. The lists returned by keys() and items() can include the same key more than once if there is a multi-valued header. The len() of a Headers object is the same as the length of its items(), which is the same as the length of the wrapped header list. In fact, the items() method just returns a copy of the wrapped header list.

Calling str() on a Headers object returns a formatted string suitable for transmission as HTTP response headers. Each header is placed on a line with its value, separated by a colon and a space. Each line is terminated by a carriage return and line feed, and the string is terminated with a blank line.

In addition to their mapping interface and formatting features, Headers objects also have the following methods for querying and adding multi-valued headers, and for adding headers with MIME parameters:

get_all(*name*)

Return a list of all the values for the named header.

The returned list will be sorted in the order they appeared in the original header list or were added to this instance, and may contain duplicates. Any fields deleted and re-inserted are always appended to the header list. If no fields exist with the given name, returns an empty list.

add_header(*name*, *value*, ***_params*)

Add a (possibly multi-valued) header, with optional MIME parameters specified via keyword arguments.

name is the header field to add. Keyword arguments can be used to set MIME parameters for the header field. Each parameter must be a string or None. Underscores in parameter names are converted to dashes, since dashes are illegal in Python identifiers, but many MIME parameter names include dashes. If the parameter value is a string, it is added to the header value parameters in the form name="value". If it is None, only the parameter name is added. (This is used for MIME parameters without a value.) Example usage:

```
h.add_header('content-disposition', 'attachment', filename='bud.gif')
```

The above will add a header that looks like this:

```
Content-Disposition: attachment; filename="bud.gif"
```

1.1.3 wsgiref.simple_server – a simple WSGI HTTP server

This module implements a simple HTTP server (based on BaseHTTPServer) that serves WSGI applications. Each server instance serves a single WSGI application on a given host and port. If you want to serve multiple applications on a single host and port, you should create a WSGI application that parses PATH_INFO to select which application to invoke for each request. (E.g., using the shift_path_info() function from wsgiref.util.)

make_server(*host, port, app* [, *server_class=WSGIServer* [, *handler_class=WSGIRequestHandler*]])

Create a new WSGI server listening on *host* and *port*, accepting connections for *app*. The return value is an instance of the supplied *server_class*, and will process requests using the specified *handler_class*. *app* must be a WSGI application object, as defined by PEP 333.

Example usage:

```
from wsgiref.simple_server import make_server, demo_app

httpd = make_server('', 8000, demo_app)
print "Serving HTTP on port 8000..."

# Respond to requests until process is killed
httpd.serve_forever()

# Alternative: serve one request, then exit
##httpd.handle_request()
```

demo_app(*environ, start_response*)

This function is a small but complete WSGI application that returns a text page containing the message "Hello world!" and a list of the key/value pairs provided in the *environ* parameter. It's useful for verifying that a WSGI server (such as `wsgiref.simple_server`) is able to run a simple WSGI application correctly.

class WSGIServer(*server_address, RequestHandlerClass*)

Create a `WSGIServer` instance. *server_address* should be a `(host,port)` tuple, and *RequestHandlerClass* should be the subclass of `BaseHTTPServer.BaseHTTPRequestHandler` that will be used to process requests.

You do not normally need to call this constructor, as the `make_server()` function can handle all the details for you.

`WSGIServer` is a subclass of `BaseHTTPServer.HTTPServer`, so all of its methods (such as `serve_forever()` and `handle_request()`) are available. `WSGIServer` also provides these WSGI-specific methods:

set_app(*application*)

Sets the callable *application* as the WSGI application that will receive requests.

get_app()

Returns the currently-set application callable.

Normally, however, you do not need to use these additional methods, as `set_app()` is normally called by `make_server()`, and the `get_app()` exists mainly for the benefit of request handler instances.

class WSGIRequestHandler(*request, client_address, server*)

Create an HTTP handler for the given *request* (i.e. a socket), *client_address* (a `(host,port)` tuple), and *server* (`WSGIServer` instance).

You do not need to create instances of this class directly; they are automatically created as needed by `WSGIServer` objects. You can, however, subclass this class and supply it as a *handler_class* to the `make_server()` function. Some possibly relevant methods for overriding in subclasses:

get_environ()

Returns a dictionary containing the WSGI environment for a request. The default implementation copies the contents of the `WSGIServer` object's `base_environ` dictionary attribute and then adds various headers derived from the HTTP request. Each call to this method should return a new dictionary containing all of the relevant CGI environment variables as specified in PEP 333.

get_stderr()

Return the object that should be used as the `wsgi.errors` stream. The default implementation just returns `sys.stderr`.

handle()
Process the HTTP request. The default implementation creates a handler instance using a wsgiref.handlers class to implement the actual WSGI application interface.

1.1.4 wsgiref.validate – WSGI conformance checker

When creating new WSGI application objects, frameworks, servers, or middleware, it can be useful to validate the new code's conformance using wsgiref.validate. This module provides a function that creates WSGI application objects that validate communications between a WSGI server or gateway and a WSGI application object, to check both sides for protocol conformance.

Note that this utility does not guarantee complete PEP 333 compliance; an absence of errors from this module does not necessarily mean that errors do not exist. However, if this module does produce an error, then it is virtually certain that either the server or application is not 100% compliant.

This module is based on the paste.lint module from Ian Bicking's "Python Paste" library.

validator(*application*)
Wrap *application* and return a new WSGI application object. The returned application will forward all requests to the original *application*, and will check that both the *application* and the server invoking it are conforming to the WSGI specification and to RFC 2616.

Any detected nonconformance results in an AssertionError being raised; note, however, that how these errors are handled is server-dependent. For example, wsgiref.simple_server and other servers based on wsgiref.handlers (that don't override the error handling methods to do something else) will simply output a message that an error has occurred, and dump the traceback to sys.stderr or some other error stream.

This wrapper may also generate output using the warnings module to indicate behaviors that are questionable but which may not actually be prohibited by PEP 333. Unless they are suppressed using Python command-line options or the warnings API, any such warnings will be written to sys.stderr (*not* wsgi.errors, unless they happen to be the same object).

1.1.5 wsgiref.handlers – server/gateway base classes

This module provides base handler classes for implementing WSGI servers and gateways. These base classes handle most of the work of communicating with a WSGI application, as long as they are given a CGI-like environment, along with input, output, and error streams.

class CGIHandler()
CGI-based invocation via sys.stdin, sys.stdout, sys.stderr and os.environ. This is useful when you have a WSGI application and want to run it as a CGI script. Simply invoke CGIHandler().run(app), where app is the WSGI application object you wish to invoke.

This class is a subclass of BaseCGIHandler that sets wsgi.run_once to true, wsgi.multithread to false, and wsgi.multiprocess to true, and always uses sys and os to obtain the necessary CGI streams and environment.

class BaseCGIHandler(*stdin, stdout, stderr, environ* [, *multithread=True* [, *multiprocess=False*]])
Similar to CGIHandler, but instead of using the sys and os modules, the CGI environment and I/O streams are specified explicitly. The *multithread* and *multiprocess* values are used to set the wsgi.multithread and wsgi.multiprocess flags for any applications run by the handler instance.

This class is a subclass of SimpleHandler intended for use with software other than HTTP "origin servers". If you are writing a gateway protocol implementation (such as CGI, FastCGI, SCGI, etc.) that uses a Status: header to send an HTTP status, you probably want to subclass this instead of SimpleHandler.

class SimpleHandler(*stdin, stdout, stderr, environ* [,*multithread=True* [, *multiprocess=False*]])
Similar to BaseCGIHandler, but designed for use with HTTP origin servers. If you are writing an HTTP

server implementation, you will probably want to subclass this instead of `BaseCGIHandler`

This class is a subclass of `BaseHandler`. It overrides the `__init__()`, `get_stdin()`, `get_stderr()`, `add_cgi_vars()`, `_write()`, and `_flush()` methods to support explicitly setting the environment and streams via the constructor. The supplied environment and streams are stored in the `stdin`, `stdout`, `stderr`, and `environ` attributes.

class `BaseHandler`()

This is an abstract base class for running WSGI applications. Each instance will handle a single HTTP request, although in principle you could create a subclass that was reusable for multiple requests.

`BaseHandler` instances have only one method intended for external use:

`run`(*app*)

Run the specified WSGI application, *app*.

All of the other `BaseHandler` methods are invoked by this method in the process of running the application, and thus exist primarily to allow customizing the process.

The following methods MUST be overridden in a subclass:

`_write`(*data*)

Buffer the string *data* for transmission to the client. It's okay if this method actually transmits the data; `BaseHandler` just separates write and flush operations for greater efficiency when the underlying system actually has such a distinction.

`_flush`()

Force buffered data to be transmitted to the client. It's okay if this method is a no-op (i.e., if `_write()` actually sends the data).

`get_stdin`()

Return an input stream object suitable for use as the `wsgi.input` of the request currently being processed.

`get_stderr`()

Return an output stream object suitable for use as the `wsgi.errors` of the request currently being processed.

`add_cgi_vars`()

Insert CGI variables for the current request into the `environ` attribute.

Here are some other methods and attributes you may wish to override. This list is only a summary, however, and does not include every method that can be overridden. You should consult the docstrings and source code for additional information before attempting to create a customized `BaseHandler` subclass.

Attributes and methods for customizing the WSGI environment:

`wsgi_multithread`

The value to be used for the `wsgi.multithread` environment variable. It defaults to true in `BaseHandler`, but may have a different default (or be set by the constructor) in the other subclasses.

`wsgi_multiprocess`

The value to be used for the `wsgi.multiprocess` environment variable. It defaults to true in `BaseHandler`, but may have a different default (or be set by the constructor) in the other subclasses.

`wsgi_run_once`

The value to be used for the `wsgi.run_once` environment variable. It defaults to false in `BaseHandler`, but `CGIHandler` sets it to true by default.

`os_environ`

The default environment variables to be included in every request's WSGI environment. By default, this is a copy of `os.environ` at the time that `wsgiref.handlers` was imported, but subclasses can either create their own at the class or instance level. Note that the dictionary should be considered read-only, since the default value is shared between multiple classes and instances.

server_software
If the origin_server attribute is set, this attribute's value is used to set the default SERVER_SOFTWARE WSGI environment variable, and also to set a default Server: header in HTTP responses. It is ignored for handlers (such as BaseCGIHandler and CGIHandler) that are not HTTP origin servers.

get_scheme()
Return the URL scheme being used for the current request. The default implementation uses the guess_scheme() function from wsgiref.util to guess whether the scheme should be "http" or "https", based on the current request's environ variables.

setup_environ()
Set the environ attribute to a fully-populated WSGI environment. The default implementation uses all of the above methods and attributes, plus the get_stdin(), get_stderr(), and add_cgi_vars() methods and the wsgi_file_wrapper attribute. It also inserts a SERVER_SOFTWARE key if not present, as long as the origin_server attribute is a true value and the server_software attribute is set.

Methods and attributes for customizing exception handling:

log_exception(*exc_info*)
Log the *exc_info* tuple in the server log. *exc_info* is a (*type*, *value*, *traceback*) tuple. The default implementation simply writes the traceback to the request's wsgi.errors stream and flushes it. Subclasses can override this method to change the format or retarget the output, mail the traceback to an administrator, or whatever other action may be deemed suitable.

traceback_limit
The maximum number of frames to include in tracebacks output by the default log_exception() method. If None, all frames are included.

error_output(*environ*, *start_response*)
This method is a WSGI application to generate an error page for the user. It is only invoked if an error occurs before headers are sent to the client.

This method can access the current error information using sys.exc_info(), and should pass that information to *start_response* when calling it (as described in the "Error Handling" section of PEP 333).

The default implementation just uses the error_status, error_headers, and error_body attributes to generate an output page. Subclasses can override this to produce more dynamic error output.

Note, however, that it's not recommended from a security perspective to spit out diagnostics to any old user; ideally, you should have to do something special to enable diagnostic output, which is why the default implementation doesn't include any.

error_status
The HTTP status used for error responses. This should be a status string as defined in PEP 333; it defaults to a 500 code and message.

error_headers
The HTTP headers used for error responses. This should be a list of WSGI response headers ((*name*, *value*) tuples), as described in PEP 333. The default list just sets the content type to text/plain.

error_body
The error response body. This should be an HTTP response body string. It defaults to the plain text, "A server error occurred. Please contact the administrator."

Methods and attributes for PEP 333's "Optional Platform-Specific File Handling" feature:

wsgi_file_wrapper
A wsgi.file_wrapper factory, or None. The default value of this attribute is the FileWrapper class from wsgiref.util.

sendfile()
Override to implement platform-specific file transmission. This method is called only if the application's return value is an instance of the class specified by the wsgi_file_wrapper attribute. It should return

a true value if it was able to successfully transmit the file, so that the default transmission code will not be executed. The default implementation of this method just returns a false value.

Miscellaneous methods and attributes:

`origin_server`
: This attribute should be set to a true value if the handler's `_write()` and `_flush()` are being used to communicate directly to the client, rather than via a CGI-like gateway protocol that wants the HTTP status in a special `Status:` header.

  This attribute's default value is true in `BaseHandler`, but false in `BaseCGIHandler` and `CGIHandler`.

`http_version`
: If `origin_server` is true, this string attribute is used to set the HTTP version of the response set to the client. It defaults to `"1.0"`.