waitress Documentation

Release 1.2.1

Pylons Project Developers

January 26, 2019

Contents

Waitress is meant to be a production-quality pure-Python WSGI server with very acceptable performance. It has no dependencies except ones which live in the Python standard library. It runs on CPython on Unix and Windows under Python 2.7+ and Python 3.4+. It is also known to run on PyPy 1.6.0 on UNIX. It supports HTTP/1.0 and HTTP/1.1.

CHAPTER 1

Extended Documentation

1.1 Usage

The following code will run waitress on port 8080 on all available IP addresses, both IPv4 and IPv6.

```
from waitress import serve
serve(wsgiapp, listen='*:8080')
```

Press `Ctrl-C` (or `Ctrl-Break` on Windows) to exit the server.

The following will run waitress on port 8080 on all available IPv4 addresses, but not IPv6.

```
from waitress import serve
serve(wsgiapp, host='0.0.0.0', port=8080)
```

By default Waitress binds to any IPv4 address on port 8080. You can omit the `host` and `port` arguments and just call `serve` with the WSGI app as a single argument:

```
from waitress import serve
serve(wsgiapp)
```

If you want to serve your application through a UNIX domain socket (to serve a downstream HTTP server/proxy such as nginx, lighttpd, and so on), call `serve` with the `unix_socket` argument:

```
from waitress import serve
serve(wsgiapp, unix_socket='/path/to/unix.sock')
```

Needless to say, this configuration won't work on Windows.

Exceptions generated by your application will be shown on the console by default. See *Access Logging* to change this.

There's an entry point for *PasteDeploy* (`egg:waitress#main`) that lets you use Waitress's WSGI gateway from a configuration file, e.g.:

```
[server:main]
use = egg:waitress#main
listen = 127.0.0.1:8080
```

Using `host` and `port` is also supported:

```
[server:main]
host = 127.0.0.1
port = 8080
```

The *PasteDeploy* syntax for UNIX domain sockets is analagous:

```
[server:main]
use = egg:waitress#main
unix_socket = /path/to/unix.sock
```

You can find more settings to tweak (arguments to `waitress.serve` or equivalent settings in PasteDeploy) in *Arguments to waitress.serve*.

Additionally, there is a command line runner called `waitress-serve`, which can be used in development and in situations where the likes of *PasteDeploy* is not necessary:

```
# Listen on both IPv4 and IPv6 on port 8041
waitress-serve --listen=*:8041 myapp:wsgifunc

# Listen on only IPv4 on port 8041
waitress-serve --port=8041 myapp:wsgifunc
```

For more information on this, see *waitress-serve*.

1.2 Access Logging

The WSGI design is modular. Waitress logs error conditions, debugging output, etc., but not web traffic. For web traffic logging, Paste provides TransLogger *middleware*. TransLogger produces logs in the Apache Combined Log Format.

1.2.1 Logging to the Console Using Python

`waitress.serve` calls `logging.basicConfig()` to set up logging to the console when the server starts up. Assuming no other logging configuration has already been done, this sets the logging default level to `logging.WARNING`. The Waitress logger will inherit the root logger's level information (it logs at level `WARNING` or above).

Waitress sends its logging output (including application exception renderings) to the Python logger object named `waitress`. You can influence the logger level and output stream using the normal Python `logging` module API. For example:

```
import logging
logger = logging.getLogger('waitress')
logger.setLevel(logging.INFO)
```

Within a PasteDeploy configuration file, you can use the normal Python `logging` module `.ini` file format to change similar Waitress logging options. For example:

```
[logger_waitress]
level = INFO
```

1.2.2 Logging to the Console Using PasteDeploy

TransLogger will automatically setup a logging handler to the console when called with no arguments. It "just works" in environments that don't configure logging. This is by virtue of its default configuration setting of `setup_console_handler = True`.

1.2.3 Logging to a File Using PasteDeploy

TransLogger does not write to files, and the Python logging system must be configured to do this. The Python class `FileHandler` logging handler can be used alongside TransLogger to create an `access.log` file similar to Apache's.

Like any standard *middleware* with a Paste entry point, TransLogger can be configured to wrap your application using `.ini` file syntax. First add a `[filter:translogger]` section, then use a `[pipeline:main]` section file to form a WSGI pipeline with both the translogger and your application in it. For instance, if you have this:

```
[app:wsgiapp]
use = egg:mypackage#wsgiapp

[server:main]
use = egg:waitress#main
host = 127.0.0.1
port = 8080
```

Add this:

```
[filter:translogger]
use = egg:Paste#translogger
setup_console_handler = False

[pipeline:main]
pipeline = translogger
           wsgiapp
```

Using PasteDeploy this way to form and serve a pipeline is equivalent to wrapping your app in a TransLogger instance via the bottom of the `main` function of your project's `__init__` file:

```
from mypackage import wsgiapp
from waitress import serve
from paste.translogger import TransLogger
serve(TransLogger(wsgiapp, setup_console_handler=False))
```

Note: TransLogger will automatically set up a logging handler to the console when called with no arguments, so it "just works" in environments that don't configure logging. Since our logging handlers are configured, we disable the automation via `setup_console_handler = False`.

With the filter in place, TransLogger's logger (named the `wsgi` logger) will propagate its log messages to the parent logger (the root logger), sending its output to the console when we request a page:

```
00:50:53,694 INFO [wsgiapp] Returning: Hello World!
                   (content-type: text/plain)
00:50:53,695 INFO [wsgi] 192.168.1.111 - - [11/Aug/2011:20:09:33 -0700] "GET /hello
HTTP/1.1" 404 - "-"
"Mozilla/5.0 (Macintosh; U; Intel Mac OS X; en-US; rv:1.8.1.6) Gecko/20070725
Firefox/2.0.0.6"
```

To direct TransLogger to an `access.log` FileHandler, we need the following to add a FileHandler (named `accesslog`) to the list of handlers, and ensure that the `wsgi` logger is configured and uses this handler accordingly:

```
# Begin logging configuration

[loggers]
keys = root, wsgiapp, wsgi

[handlers]
keys = console, accesslog

[logger_wsgi]
level = INFO
handlers = accesslog
qualname = wsgi
propagate = 0

[handler_accesslog]
class = FileHandler
args = ('%(here)s/access.log','a')
level = INFO
formatter = generic
```

As mentioned above, non-root loggers by default propagate their log records to the root logger's handlers (currently the console handler). Setting `propagate` to `0` (`False`) here disables this; so the `wsgi` logger directs its records only to the `accesslog` handler.

Finally, there's no need to use the `generic` formatter with TransLogger, as TransLogger itself provides all the information we need. We'll use a formatter that passes-through the log messages as is. Add a new formatter called `accesslog` by including the following in your configuration file:

```
[formatters]
keys = generic, accesslog

[formatter_accesslog]
format = %(message)s
```

Finally alter the existing configuration to wire this new `accesslog` formatter into the FileHandler:

```
[handler_accesslog]
class = FileHandler
args = ('%(here)s/access.log','a')
level = INFO
formatter = accesslog
```

1.3 Using Behind a Reverse Proxy

Often people will set up "pure Python" web servers behind reverse proxies, especially if they need TLS support (Waitress does not natively support TLS). Even if you don't need TLS support, it's not uncommon to see Waitress and other pure-Python web servers set up to only handle requests behind a reverse proxy; these proxies often have lots of useful deployment knobs.

If you're using Waitress behind a reverse proxy, you'll almost always want your reverse proxy to pass along the `Host` header sent by the client to Waitress, in either case, as it will be used by most applications to generate correct URLs. You may also use the proxy headers if passing `Host` directly is not possible, or there are multiple proxies involved.

For example, when using nginx as a reverse proxy, you might add the following lines in a `location` section.

```
proxy_set_header        Host $host;
```

The Apache directive named `ProxyPreserveHost` does something similar when used as a reverse proxy.

Unfortunately, even if you pass the `Host` header, the Host header does not contain enough information to regenerate the original URL sent by the client. For example, if your reverse proxy accepts HTTPS requests (and therefore URLs which start with `https://`), the URLs generated by your application when used behind a reverse proxy served by Waitress might inappropriately be `http://foo` rather than `https://foo`. To fix this, you'll want to change the `wsgi.url_scheme` in the WSGI environment before it reaches your application. You can do this in one of three ways:

1. You can pass a `url_scheme` configuration variable to the `waitress.serve` function.
2. You can pass certain well known proxy headers from your proxy server and use waitress's `trusted_proxy` support to automatically configure the WSGI environment.

1.3.1 Using `url_scheme` to set `wsgi.url_scheme`

You can have the Waitress server use the `https` url scheme by default.:

```
from waitress import serve
serve(wsgiapp, listen='0.0.0.0:8080', url_scheme='https')
```

This works if all URLs generated by your application should use the `https` scheme.

1.3.2 Passing the proxy headers to setup the WSGI environment

If your proxy accepts both HTTP and HTTPS URLs, and you want your application to generate the appropriate url based on the incoming scheme, you'll want to pass waitress `X-Forwarded-Proto`, however Waitress is also able to update the environment using `X-Forwarded-Proto`, `X-Forwarded-For`, `X-Forwarded-Host`, and `X-Forwarded-Port`:

```
proxy_set_header X-Forwarded-Proto $scheme;
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
proxy_set_header X-Forwarded-Host $host:$server_port;
proxy_set_header X-Forwarded-Port $server_port;
```

when using Apache, `mod_proxy` automatically forwards the following headers:

```
X-Forwarded-For
X-Forwarded-Host
X-Forwarded-Server
```

You will also want to add to Apache:

```
RequestHeader set X-Forwarded-Proto https
```

Configure waitress's `trusted_proxy_headers` as appropriate:

```
trusted_proxy_headers = "x-forwarded-for x-forwarded-host x-forwarded-proto x-
↪forwarded-port"
```

At this point waitress will set up the WSGI environment using the information specified in the trusted proxy headers. This will setup the following variables:

```
HTTP_HOST
SERVER_NAME
SERVER_PORT
REMOTE_ADDR
REMOTE_PORT (if available)
wsgi.url_scheme
```

Waitress also has support for the Forwarded (RFC7239) HTTP header which is better defined than the ad-hoc `X-Forwarded-*`, however support is not nearly as widespread yet. `Forwarded` supports similar functionality as the different individual headers, and is mutually exclusive to using the `X-Forwarded-*` headers.

To configure waitress to use the `Forwarded` header, set:

```
trusted_proxy_headers = "forwarded"
```

Note: You must also configure the Waitress server's `trusted_proxy` to contain the IP address of the proxy.

1.3.3 Using `url_prefix` to influence `SCRIPT_NAME` and `PATH_INFO`

You can have the Waitress server use a particular url prefix by default for all URLs generated by downstream applications that take `SCRIPT_NAME` into account.:

```
from waitress import serve
serve(wsgiapp, listen='0.0.0.0:8080', url_prefix='/foo')
```

Setting this to any value except the empty string will cause the WSGI `SCRIPT_NAME` value to be that value, minus any trailing slashes you add, and it will cause the `PATH_INFO` of any request which is prefixed with this value to be stripped of the prefix. This is useful in proxying scenarios where you wish to forward all traffic to a Waitress server but need URLs generated by downstream applications to be prefixed with a particular path segment.

1.4 Design

Waitress uses a combination of asynchronous and synchronous code to do its job. It handles I/O to and from clients using the *wasyncore*, which is *asyncore* vendored into Waitress. It services requests via threads.

Note: *asyncore* has been deprecated since Python 3.6. Work continues on its inevitable removal from the Python standard library. Its recommended replacement is `asyncio`.

Although *asyncore* has been vendored into Waitress as *wasyncore*, you may see references to "asyncore" in this documentation's code examples and API. The terms are effectively the same and may be used interchangeably.

The *wasyncore* module:

- Uses the `select.select` function to wait for connections from clients and determine if a connected client is ready to receive output.
- Creates a channel whenever a new connection is made to the server.
- Executes methods of a channel whenever it believes data can be read from or written to the channel.

A "channel" is created for each connection from a client to the server. The channel handles all requests over the same connection from that client. A channel will handle some number of requests during its lifetime: zero to how ever many HTTP requests are sent to the server by the client over a single connection. For example, an HTTP/1.1 client may issue a theoretically infinite number of requests over the same connection; each of these will be handled by the same channel. An HTTP/1.0 client without a "Connection: keep-alive" header will request usually only one over a single TCP connection, however, and when the request has completed, the client disconnects and reconnects (which will create another channel). When the connection related to a channel is closed, the channel is destroyed and garbage collected.

When a channel determines the client has sent at least one full valid HTTP request, it schedules a "task" with a "thread dispatcher". The thread dispatcher maintains a fixed pool of worker threads available to do client work (by default, 4 threads). If a worker thread is available when a task is scheduled, the worker thread runs the task. The task has access to the channel, and can write back to the channel's output buffer. When all worker threads are in use, scheduled tasks will wait in a queue for a worker thread to become available.

I/O is always done asynchronously (by *wasyncore*) in the main thread. Worker threads never do any I/O. This means that

1. a large number of clients can be connected to the server at once, and
2. worker threads will never be hung up trying to send data to a slow client.

No attempt is made to kill a "hung thread". It's assumed that when a task (application logic) starts that it will eventually complete. If for some reason WSGI application logic never completes and spins forever, the worker thread related to that WSGI application will be consumed "forever", and if enough worker threads are consumed like this, the server will stop responding entirely.

Periodic maintenance is done by the main thread (the thread handling I/O). If a channel hasn't sent or received any data in a while, the channel's connection is closed, and the channel is destroyed.

1.5 Differences from `zope.server`

- Has no non-stdlib dependencies.
- No support for non-WSGI servers (no FTP, plain-HTTP, etc); refactorings and slight interface changes as a result. Non-WSGI-supporting code removed.
- Slight cleanup in the way application response headers are handled (no more "accumulated headers").
- Supports the HTTP 1.1 "expect/continue" mechanism (required by WSGI spec).
- Calls "close()" on the app_iter object returned by the WSGI application.
- Allows trusted proxies to override `wsgi.url_scheme` for particular requests by supplying the `X_FORWARDED_PROTO` header.
- Supports an explicit `wsgi.url_scheme` parameter for ease of deployment behind SSL proxies.

- Different adjustment defaults (less conservative).
- Python 3 compatible.
- More test coverage (unit tests added, functional tests refactored and more added).
- Supports convenience `waitress.serve` function (e.g. `from waitress import serve; serve(app)` and convenience `server.run()` function.
- Returns a "real" write method from start_response.
- Provides a getsockname method of the server FBO figuring out which port the server is listening on when it's bound to port 0.
- Warns when app_iter bytestream numbytes less than or greater than specified Content-Length.
- Set content-length header if len(app_iter) == 1 and none provided.
- Raise an exception if start_response isnt called before any body write.
- channel.write does not accept non-byte-sequences.
- Put maintenance check on server rather than channel to avoid a class of DOS.
- wsgi.multiprocess set (correctly) to False.
- Ensures header total can not exceed a maximum size.
- Ensures body total can not exceed a maximum size.
- Broken chunked encoding request bodies don't crash the server.
- Handles keepalive/pipelining properly (no out of order responses, no premature channel closes).
- Send a 500 error to the client when a task raises an uncaught exception (with optional traceback rendering via "expose_traceback" adjustment).
- Supports HTTP/1.1 chunked responses when application doesn't set a Content-Length header.
- Dont hang a thread up trying to send data to slow clients.
- Supports `wsgi.file_wrapper` protocol.

1.6 `waitress` API

serve(*app*, *listen='0.0.0.0:8080'*, *unix_socket=None*, *unix_socket_perms='600'*, *threads=4*, *url_scheme='http'*, *url_prefix=''*, *ident='waitress'*, *backlog=1204*, *recv_bytes=8192*, *send_bytes=18000*, *outbuf_overflow=104856*, *inbuf_overflow=52488*, *connection_limit=1000*, *cleanup_interval=30*, *channel_timeout=120*, *log_socket_errors=True*, *max_request_header_size=262144*, *max_request_body_size=1073741824*, *expose_tracebacks=False*)

See *Arguments to waitress.serve* for more information.

1.7 Arguments to `waitress.serve`

Here are the arguments you can pass to the *waitress.serve'* function or use in *PasteDeploy* configuration (interchangeably):

host Hostname or IP address (string) on which to listen, default `0.0.0.0`, which means "all IP addresses on this host".

> **Warning:** May not be used with `listen`

port TCP port (integer) on which to listen, default `8080`

> **Warning:** May not be used with `listen`

listen Tell waitress to listen on combinations of `host:port` arguments. Combinations should be a quoted, space-delimited list, as in the following examples.

```
listen="127.0.0.1:8080 [::1]:8080"
listen="*:8080 *:6543"
```

A wildcard for the hostname is also supported and will bind to both IPv4/IPv6 depending on whether they are enabled or disabled.

IPv6 IP addresses are supported by surrounding the IP address with brackets.

New in version 1.0.

ipv4 Enable or disable IPv4 (boolean)

ipv6 Enable or disable IPv6 (boolean)

unix_socket Path of Unix socket (string). If a socket path is specified, a Unix domain socket is made instead of the usual inet domain socket.

Not available on Windows.

Default: `None`

unix_socket_perms Octal permissions to use for the Unix domain socket (string). Only used if `unix_socket` is not `None`.

Default: `'600'`

sockets A list of sockets. The sockets can be either Internet or UNIX sockets and have to be bound. Internet and UNIX sockets cannot be mixed. If the socket list is not empty, waitress creates one server for each socket.

Default: `[]`

New in version 1.1.1.

> **Warning:** May not be used with `listen`, `host`, `port` or `unix_socket`

threads The number of threads used to process application logic (integer).

Default: `4`

trusted_proxy IP address of a remote peer allowed to override various WSGI environment variables using proxy headers.

For unix sockets, set this value to `localhost` instead of an IP address.

Default: `None`

trusted_proxy_count How many proxies we trust when chained. For example,

```
X-Forwarded-For:  192.0.2.1, "[2001:db8::1]"
```

or

```
Forwarded:  for=192.0.2.1, For="[2001:db8::1]"
```

means there were (potentially), two proxies involved. If we know there is only 1 valid proxy, then that initial IP address "192.0.2.1" is not trusted and we completely ignore it.

If there are two trusted proxies in the path, this value should be set to 2. If there are more proxies, this value should be set higher.

Default: `1`

New in version 1.2.0.

trusted_proxy_headers Which of the proxy headers should we trust, this is a set where you either specify "forwarded" or one or more of "x-forwarded-host", "x-forwarded-for", "x-forwarded-proto", "x-forwarded-port", "x-forwarded-by".

This list of trusted headers is used when `trusted_proxy` is set and will allow waitress to modify the WSGI environment using the values provided by the proxy.

New in version 1.2.0.

> **Warning:** If `trusted_proxy` is set, the default is `x-forwarded-proto` to match older versions of Waitress. Users should explicitly opt-in by selecting the headers to be trusted as future versions of waitress will use an empty default.

> **Warning:** It is an error to set this value without setting `trusted_proxy`.

log_untrusted_proxy_headers Should waitress log warning messages about proxy headers that are being sent from upstream that are not trusted by `trusted_proxy_headers` but are being cleared due to `clear_untrusted_proxy_headers`?

This may be useful for debugging if you expect your upstream proxy server to only send specific headers.

Default: `False`

New in version 1.2.0.

> **Warning:** It is a no-op to set this value without also setting `clear_untrusted_proxy_headers` and `trusted_proxy`

clear_untrusted_proxy_headers This tells Waitress to remove any untrusted proxy headers ("Forwarded", "X-Forwared-For", "X-Forwarded-By", "X-Forwarded-Host", "X-Forwarded-Port", "X-Forwarded-Proto") not explicitly allowed by `trusted_proxy_headers`.

Default: `False`

New in version 1.2.0.

> **Warning:** The default value is set to `False` for backwards compatibility. In future versions of Waitress this default will be changed to `True`. Warnings will be raised unless the user explicitly provides a value for this option, allowing the user to opt-in to the new safety features automatically.

> **Warning:** It is an error to set this value without setting `trusted_proxy`.

url_scheme The value of `wsgi.url_scheme` in the environ. This can be overridden per-request by the value of the `X_FORWARDED_PROTO` header, but only if the client address matches `trusted_proxy`.

Default: `http`

ident Server identity (string) used in "Server:" header in responses.

Default: `waitress`

backlog The value waitress passes to pass to `socket.listen()` (integer). This is the maximum number of incoming TCP connections that will wait in an OS queue for an available channel. From listen(1): "If a connection request arrives when the queue is full, the client may receive an error with an indication of ECONNREFUSED or, if the underlying protocol supports retransmission, the request may be ignored so that a later reattempt at connection succeeds."

Default: `1024`

recv_bytes The argument waitress passes to `socket.recv()` (integer).

Default: `8192`

send_bytes The number of bytes to send to `socket.send()` (integer). Multiples of 9000 should avoid partly-filled TCP packets, but don't set this larger than the TCP write buffer size. In Linux, `/proc/sys/net/ipv4/tcp_wmem` controls the minimum, default, and maximum sizes of TCP write buffers.

Default: `18000`

outbuf_overflow A tempfile should be created if the pending output is larger than outbuf_overflow, which is measured in bytes. The default is conservative.

Default: `1048576` (1MB)

inbuf_overflow A tempfile should be created if the pending input is larger than inbuf_overflow, which is measured in bytes. The default is conservative.

Default: `524288` (512K)

connection_limit Stop creating new channels if too many are already active (integer). Each channel consumes at least one file descriptor, and, depending on the input and output body sizes, potentially up to three, plus whatever file descriptors your application logic happens to open. The default is conservative, but you may need to increase the number of file descriptors available to the Waitress process on most platforms in order to safely change it (see `ulimit -a` "open files" setting). Note that this doesn't control the maximum number of TCP connections that can be waiting for processing; the `backlog` argument controls that.

Default: `100`

cleanup_interval Minimum seconds between cleaning up inactive channels (integer). See also `channel_timeout`.

Default: `30`

channel_timeout Maximum seconds to leave an inactive connection open (integer). "Inactive" is defined as "has received no data from a client and has sent no data to a client".

Default: `120`

log_socket_errors Set to `False` to not log premature client disconnect tracebacks.

Default: `True`

max_request_header_size Maximum number of bytes of all request headers combined (integer).

Default: `262144` (256K)

max_request_body_size Maximum number of bytes in request body (integer).

Default: `1073741824` (1GB)

expose_tracebacks Set to `True` to expose tracebacks of unhandled exceptions to client.

Default: `False`

asyncore_loop_timeout The `timeout` value (seconds) passed to `asyncore.loop` to run the mainloop.

Default: `1`

New in version 0.8.3.

asyncore_use_poll Set to `True` to switch from using `select()` to `poll()` in `asyncore.loop`. By default `asyncore.loop()` uses `select()` which has a limit of 1024 file descriptors. `select()` and `poll()` provide basically the same functionality, but `poll()` doesn't have the file descriptors limit.

Default: `False`

New in version 0.8.6.

url_prefix String: the value used as the WSGI `SCRIPT_NAME` value. Setting this to anything except the empty string will cause the WSGI `SCRIPT_NAME` value to be the value passed minus any trailing slashes you add, and it will cause the `PATH_INFO` of any request which is prefixed with this value to be stripped of the prefix.

Default: `''`

1.8 Support for `wsgi.file_wrapper`

Waitress supports the Python Web Server Gateway Interface v1.0 as specified in **PEP 3333**. Here's a usage example:

```
import os

here = os.path.dirname(os.path.abspath(__file__))

def myapp(environ, start_response):
    f = open(os.path.join(here, 'myphoto.jpg'), 'rb')
    headers = [('Content-Type', 'image/jpeg')]
    start_response(
        '200 OK',
        headers
        )
    return environ['wsgi.file_wrapper'](f, 32768)
```

The file wrapper constructor is accessed via `environ['wsgi.file_wrapper']`. The signature of the file wrapper constructor is `(filelike_object, block_size)`. Both arguments must be passed as positional (not keyword) arguments. The result of creating a file wrapper should be **returned** as the `app_iter` from a WSGI application.

The object passed as `filelike_object` to the wrapper must be a file-like object which supports *at least* the `read()` method, and the `read()` method must support an optional size hint argument and the `read()` method *must* return **bytes** objects (never unicode). It *should* support the `seek()` and `tell()` methods. If it does not, normal iteration over the `filelike_object` using the provided `block_size` is used (and copying is done, negating any benefit of the file wrapper). It *should* support a `close()` method.

The specified `block_size` argument to the file wrapper constructor will be used only when the `filelike_object` doesn't support `seek` and/or `tell` methods. Waitress needs to use normal iteration to serve

the file in this degenerate case (as per the WSGI pec), and this block size will be used as the iteration chunk size. The `block_size` argument is optional; if it is not passed, a default value `32768` is used.

Waitress will set a `Content-Length` header on behalf of an application when a file wrapper with a sufficiently file-like object is used if the application hasn't already set one.

The machinery which handles a file wrapper currently doesn't do anything particularly special using fancy system calls (it doesn't use `sendfile` for example); using it currently just prevents the system from needing to copy data to a temporary buffer in order to send it to the client. No copying of data is done when a WSGI app returns a file wrapper that wraps a sufficiently file-like object. It may do something fancier in the future.

1.9 waitress-serve

New in version 0.8.4: Waitress comes bundled with a thin command-line wrapper around the `waitress.serve` function called `waitress-serve`. This is useful for development, and in production situations where serving of static assets is delegated to a reverse proxy, such as nginx or Apache.

`waitress-serve` takes the very same *arguments* as the `waitress.serve` function, but where the function's arguments have underscores, `waitress-serve` uses hyphens. Thus:

```
import myapp

waitress.serve(myapp.wsgifunc, port=8041, url_scheme='https')
```

Is equivalent to:

```
waitress-serve --port=8041 --url-scheme=https myapp:wsgifunc
```

The full argument list is *given below*.

Boolean arguments are represented by flags. If you wish to explicitly set a flag, simply use it by its name. Thus the flag:

```
--expose-tracebacks
```

Is equivalent to passing `expose_tracebacks=True` to `waitress.serve`.

All flags have a negative equivalent. These are prefixed with `no-`; thus using the flag:

```
--no-expose-tracebacks
```

Is equivalent to passing `expose_tracebacks=False` to `waitress.serve`.

If at any time you want the full argument list, use the `--help` flag.

Applications are specified similarly to PasteDeploy, where the format is `myapp.mymodule:wsgifunc`. As some application frameworks use application objects, you can use dots to resolve attributes like so: `myapp.mymodule:appobj.wsgifunc`.

A number of frameworks, *web.py* being an example, have factory methods on their application objects that return usable WSGI functions when called. For cases like these, `waitress-serve` has the `--call` flag. Thus:

```
waitress-serve --call myapp.mymodule.app.wsgi_factory
```

Would load the `myapp.mymodule` module, and call `app.wsgi_factory` to get a WSGI application function to be passed to `waitress.server`.

Note: As of 0.8.6, the current directory is automatically included on `sys.path`.

1.9.1 Invocation

Usage:

```
waitress-serve [OPTS] MODULE:OBJECT
```

Common options:

--help Show this information.

--call Call the given object to get the WSGI application.

--host=ADDR Hostname or IP address on which to listen, default is '0.0.0.0', which means "all IP addresses on this host".

--port=PORT TCP port on which to listen, default is '8080'

--listen=host:port Tell waitress to listen on an ip port combination.

Example:

–listen=127.0.0.1:8080 –listen=[::1]:8080 –listen=*:8080

This option may be used multiple times to listen on multipe sockets. A wildcard for the hostname is also supported and will bind to both IPv4/IPv6 depending on whether they are enabled or disabled.

--[no-]ipv4 Toggle on/off IPv4 support.

This affects wildcard matching when listening on a wildcard address/port combination.

--[no-]ipv6 Toggle on/off IPv6 support.

This affects wildcard matching when listening on a wildcard address/port combination.

--unix-socket=PATH Path of Unix socket. If a socket path is specified, a Unix domain socket is made instead of the usual inet domain socket.

Not available on Windows.

--unix-socket-perms=PERMS Octal permissions to use for the Unix domain socket, default is '600'.

--url-scheme=STR Default `wsgi.url_scheme` value, default is 'http'.

--url-prefix=STR The `SCRIPT_NAME` WSGI environment value. Setting this to anything except the empty string will cause the WSGI `SCRIPT_NAME` value to be the value passed minus any trailing slashes you add, and it will cause the `PATH_INFO` of any request which is prefixed with this value to be stripped of the prefix. Default is the empty string.

--ident=STR Server identity used in the 'Server' header in responses. Default is 'waitress'.

Tuning options:

--threads=INT Number of threads used to process application logic, default is 4.

--backlog=INT Connection backlog for the server. Default is 1024.

--recv-bytes=INT Number of bytes to request when calling `socket.recv()`. Default is 8192.

--send-bytes=INT Number of bytes to send to socket.send(). Default is 18000. Multiples of 9000 should avoid partly-filled TCP packets.

--outbuf-overflow=INT A temporary file should be created if the pending output is larger than this. Default is 1048576 (1MB).

--inbuf-overflow=INT A temporary file should be created if the pending input is larger than this. Default is 524288 (512KB).

--connection-limit=INT Stop creating new channels if too many are already active. Default is 100.

--cleanup-interval=INT Minimum seconds between cleaning up inactive channels. Default is 30. See `--channel-timeout`.

--channel-timeout=INT Maximum number of seconds to leave inactive connections open. Default is 120. 'Inactive' is defined as 'has received no data from the client and has sent no data to the client'.

--[no-]log-socket-errors Toggle whether premature client disconnect tracebacks ought to be logged. On by default.

--max-request-header-size=INT Maximum size of all request headers combined. Default is 262144 (256KB).

--max-request-body-size=INT Maximum size of request body. Default is 1073741824 (1GB).

--[no-]expose-tracebacks Toggle whether to expose tracebacks of unhandled exceptions to the client. Off by default.

--asyncore-loop-timeout=INT The timeout value in seconds passed to `asyncore.loop()`. Default is 1.

--asyncore-use-poll The use_poll argument passed to `asyncore.loop()`. Helps overcome open file descriptors limit. Default is False.

1.10 Socket Activation

While waitress does not support the various implementations of socket activation, for example using systemd or launchd, it is prepared to receive pre-bound sockets from init systems, process and socket managers, or other launchers that can provide pre-bound sockets.

The following shows a code example starting waitress with two pre-bound Internet sockets.

```
import socket
import waitress


def app(environ, start_response):
    content_length = environ.get('CONTENT_LENGTH', None)
    if content_length is not None:
        content_length = int(content_length)
    body = environ['wsgi.input'].read(content_length)
    content_length = str(len(body))
    start_response(
        '200 OK',
        [('Content-Length', content_length), ('Content-Type', 'text/plain')]
    )
    return [body]


if __name__ == '__main__':
    sockets = [
        socket.socket(socket.AF_INET, socket.SOCK_STREAM),
        socket.socket(socket.AF_INET, socket.SOCK_STREAM)]
```

(continues on next page)

(continued from previous page)

```
    sockets[0].bind(('127.0.0.1', 8080))
    sockets[1].bind(('127.0.0.1', 9090))
    waitress.serve(app, sockets=sockets)
    for socket in sockets:
        socket.close()
```

Generally, to implement socket activation for a given init system, a wrapper script uses the init system specific libraries to retrieve the sockets from the init system. Afterwards it starts waitress, passing the sockets with the parameter `sockets`. Note that the sockets have to be bound, which all init systems supporting socket activation do.

1.11 Glossary

asyncore A Python standard library module for asynchronous communications. See `asyncore`.

> Changed in version 1.2.0: Waitress has now "vendored" `asyncore` into itself as `waitress.wasyncore`. This is to cope with the eventuality that `asyncore` will be removed from the Python standard library in Python 3.8 or so.

middleware *Middleware* is a *WSGI* concept. It is a WSGI component that acts both as a server and an application. Interesting uses for middleware exist, such as caching, content-transport encoding, and other functions. See WSGI.org or PyPI to find middleware for your application.

PasteDeploy A system for configuration of WSGI web components in declarative `.ini` format. See https://docs.pylonsproject.org/projects/pastedeploy/en/latest/.

wasyncore Changed in version 1.2.0: Waitress has now "vendored" *asyncore* into itself as `waitress.wasyncore`. This is to cope with the eventuality that `asyncore` will be removed from the Python standard library in Python 3.8 or so.

WSGI Web Server Gateway Interface. This is a Python standard for connecting web applications to web servers, similar to the concept of Java Servlets. Waitress requires that your application be served as a WSGI application.

CHAPTER 2

Change History

CHAPTER 3

1.2.1 (2019-01-25)

3.1 Bugfixes

- When given an IPv6 address in `X-Forwarded-For` or `Forwarded for=` waitress was placing the IP address in `REMOTE_ADDR` with brackets: `[2001:db8::0]`, this does not match the requirements in the CGI spec which `REMOTE_ADDR` was lifted from. Waitress will now place the bare IPv6 address in `REMOTE_ADDR`: `2001:db8::0`. See https://github.com/Pylons/waitress/pull/232 and https://github.com/Pylons/waitress/issues/230

CHAPTER 4

1.2.0 (2019-01-15)

No changes since the last beta release. Enjoy Waitress!

CHAPTER 5

1.2.0b3 (2019-01-07)

5.1 Bugfixes

- Modified `clear_untrusted_proxy_headers` to be usable without a `trusted_proxy`. https://github.com/Pylons/waitress/pull/228
- Modified `trusted_proxy_count` to error when used without a `trusted_proxy`. https://github.com/Pylons/waitress/pull/228

CHAPTER 6

1.2.0b2 (2019-02-02)

6.1 Bugfixes

- Fixed logic to no longer warn on writes where the output is required to have a body but there may not be any data to be written. Solves issue posted on the Pylons Project mailing list with 1.2.0b1.

CHAPTER 7

1.2.0b1 (2018-12-31)

Happy New Year!

7.1 Features

- Setting the `trusted_proxy` setting to `'*'` (wildcard) will allow all upstreams to be considered trusted proxies, thereby allowing services behind Cloudflare/ELBs to function correctly whereby there may not be a singular IP address that requests are received from.

  Using this setting is potentially dangerous if your server is also available from anywhere on the internet, and further protections should be used to lock down access to Waitress. See https://github.com/Pylons/waitress/pull/224

- Waitress has increased its support of the X-Forwarded-* headers and includes Forwarded (RFC7239) support. This may be used to allow proxy servers to influence the WSGI environment. See https://github.com/Pylons/waitress/pull/209

  This also provides a new security feature when using Waitress behind a proxy in that it is possible to remove untrusted proxy headers thereby making sure that downstream WSGI applications don't accidentally use those proxy headers to make security decisions.

  The documentation has more information, see the following new arguments:

  - trusted_proxy_count
  - trusted_proxy_headers
  - clear_untrusted_proxy_headers
  - log_untrusted_proxy_headers (useful for debugging)

  Be aware that the defaults for these are currently backwards compatible with older versions of Waitress, this will change in a future release of waitress. If you expect to need this behaviour please explicitly set these variables in your configuration, or pin this version of waitress.

  Documentation: https://docs.pylonsproject.org/projects/waitress/en/latest/reverse-proxy.html

- Waitress can now accept a list of sockets that are already pre-bound rather than creating its own to allow for socket activation. Support for init systems/other systems that create said activated sockets is not included. See https://github.com/Pylons/waitress/pull/215
- Server header can be omitted by specifying `ident=None` or `ident=''`. See https://github.com/Pylons/waitress/pull/187

7.2 Bugfixes

- Waitress will no longer send Transfer-Encoding or Content-Length for 1xx, 204, or 304 responses, and will completely ignore any message body sent by the WSGI application, making sure to follow the HTTP standard. See https://github.com/Pylons/waitress/pull/166, https://github.com/Pylons/waitress/issues/165, https://github.com/Pylons/waitress/issues/152, and https://github.com/Pylons/waitress/pull/202

7.3 Compatibility

- Waitress has now "vendored" asyncore into itself as `waitress.wasyncore`. This is to cope with the eventuality that asyncore will be removed from the Python standard library in 3.8 or so.

7.4 Documentation

- Bring in documentation of paste.translogger from Pyramid. Reorganize and clean up documentation. See https://github.com/Pylons/waitress/pull/205 https://github.com/Pylons/waitress/pull/70 https://github.com/Pylons/waitress/pull/206

CHAPTER 8

1.1.0 (2017-10-10)

8.1 Features

- Waitress now has a __main__ and thus may be called with `python -mwaitress`

8.2 Bugfixes

- Waitress no longer allows lowercase HTTP verbs. This change was made to fall in line with most HTTP servers. See https://github.com/Pylons/waitress/pull/170
- When receiving non-ascii bytes in the request URL, waitress will no longer abruptly close the connection, instead returning a 400 Bad Request. See https://github.com/Pylons/waitress/pull/162 and https://github.com/Pylons/waitress/issues/64

CHAPTER 9

1.0.2 (2017-02-04)

9.1 Features

- Python 3.6 is now officially supported in Waitress

9.2 Bugfixes

- Add a work-around for libc issue on Linux not following the documented standards. If getnameinfo() fails because of DNS not being available it should return the IP address instead of the reverse DNS entry, however instead getnameinfo() raises. We catch this, and ask getnameinfo() for the same information again, explicitly asking for IP address instead of reverse DNS hostname. See https://github.com/Pylons/waitress/issues/149 and https://github.com/Pylons/waitress/pull/153

CHAPTER 10

1.0.1 (2016-10-22)

10.1 Bugfixes

- IPv6 support on Windows was broken due to missing constants in the socket module. This has been resolved by setting the constants on Windows if they are missing. See https://github.com/Pylons/waitress/issues/138
- A ValueError was raised on Windows when passing a string for the port, on Windows in Python 2 using service names instead of port numbers doesn't work with *getaddrinfo*. This has been resolved by attempting to convert the port number to an integer, if that fails a ValueError will be raised. See https://github.com/Pylons/waitress/issues/139

CHAPTER 11

1.0.0 (2016-08-31)

11.1 Bugfixes

- Removed *AI_ADDRCONFIG* from the call to *getaddrinfo*, this resolves an issue whereby *getaddrinfo* wouldn't return any addresses to *bind* to on hosts where there is no internet connection but localhost is requested to be bound to. See https://github.com/Pylons/waitress/issues/131 for more information.

11.2 Deprecations

- Python 2.6 is no longer supported.

11.3 Features

- IPv6 support
- Waitress is now able to listen on multiple sockets, including IPv4 and IPv6. Instead of passing in a host/port combination you now provide waitress with a space delineated list, and it will create as many sockets as required.

```
from waitress import serve
serve(wsgiapp, listen='0.0.0.0:8080 [::]:9090 *:6543')
```

11.4 Security

- Waitress will now drop HTTP headers that contain an underscore in the key when received from a client. This is to stop any possible underscore/dash conflation that may lead to security issues. See https://github.com/Pylons/waitress/pull/80 and https://www.djangoproject.com/weblog/2015/jan/13/security/

CHAPTER 12

0.9.0 (2016-04-15)

12.1 Deprecations

- Python 3.2 is no longer supported by Waitress.
- Python 2.6 will no longer be supported by Waitress in future releases.

12.2 Security/Protections

- Building on the changes made in pull request 117, add in checking for line feed/carriage return HTTP Response Splitting in the status line, as well as the key of a header. See https://github.com/Pylons/waitress/pull/124 and https://github.com/Pylons/waitress/issues/122.
- Waitress will no longer accept headers or status lines with newline/carriage returns in them, thereby disallowing HTTP Response Splitting. See https://github.com/Pylons/waitress/issues/117 for more information, as well as https://www.owasp.org/index.php/HTTP_Response_Splitting.

12.3 Bugfixes

- FileBasedBuffer and more important ReadOnlyFileBasedBuffer no longer report False when tested with bool(), instead always returning True, and becoming more iterator like. See: https://github.com/Pylons/waitress/pull/82 and https://github.com/Pylons/waitress/issues/76
- Call prune() on the output buffer at the end of a request so that it doesn't continue to grow without bounds. See https://github.com/Pylons/waitress/issues/111 for more information.

CHAPTER 13

0.8.10 (2015-09-02)

- Add support for Python 3.4, 3.5b2, and PyPy3.
- Use a nonglobal asyncore socket map by default, trying to prevent conflicts with apps and libs that use the asyncore global socket map ala https://github.com/Pylons/waitress/issues/63. You can get the old use-global-socket-map behavior back by passing `asyncore.socket_map` to the `create_server` function as the `map` argument.
- Waitress violated PEP 3333 with respect to reraising an exception when `start_response` was called with an `exc_info` argument. It would reraise the exception even if no data had been sent to the client. It now only reraises the exception if data has actually been sent to the client. See https://github.com/Pylons/waitress/pull/52 and https://github.com/Pylons/waitress/issues/51
- Add a `docs` section to tox.ini that, when run, ensures docs can be built.
- If an `application` value of `None` is supplied to the `create_server` constructor function, a ValueError is now raised eagerly instead of an error occuring during runtime. See https://github.com/Pylons/waitress/pull/60
- Fix parsing of multi-line (folded) headers. See https://github.com/Pylons/waitress/issues/53 and https://github.com/Pylons/waitress/pull/90
- Switch from the low level Python thread/_thread module to the threading module.
- Improved exception information should module import go awry.

CHAPTER 14

0.8.9 (2014-05-16)

- Fix tests under Windows. NB: to run tests under Windows, you cannot run "setup.py test" or "setup.py nosetests". Instead you must run `python.exe -c "import nose; nose.main()"`. If you try to run the tests using the normal method under Windows, each subprocess created by the test suite will attempt to run the test suite again. See https://github.com/nose-devs/nose/issues/407 for more information.
- Give the WSGI app_iter generated when `wsgi.file_wrapper` is used (ReadOnlyFileBasedBuffer) a `close` method. Do not call `close` on an instance of such a class when it's used as a WSGI app_iter, however. This is part of a fix which prevents a leakage of file descriptors; the other part of the fix was in WebOb (https://github.com/Pylons/webob/commit/951a41ce57bd853947f842028bccb500bd5237da).
- Allow trusted proxies to override `wsgi.url_scheme` via a request header, `X_FORWARDED_PROTO`. Allows proxies which serve mixed HTTP / HTTPS requests to control signal which are served as HTTPS. See https://github.com/Pylons/waitress/pull/42.

CHAPTER 15

0.8.8 (2013-11-30)

- Fix some cases where the creation of extremely large output buffers (greater than 2GB, suspected to be buffers added via `wsgi.file_wrapper`) might cause an OverflowError on Python 2. See https://github.com/Pylons/waitress/issues/47.
- When the `url_prefix` adjustment starts with more than one slash, all slashes except one will be stripped from its beginning. This differs from older behavior where more than one leading slash would be preserved in `url_prefix`.
- If a client somehow manages to send an empty path, we no longer convert the empty path to a single slash in `PATH_INFO`. Instead, the path remains empty. According to RFC 2616 section "5.1.2 Request-URI", the scenario of a client sending an empty path is actually not possible because the request URI portion cannot be empty.
- If the `url_prefix` adjustment matches the request path exactly, we now compute `SCRIPT_NAME` and `PATH_INFO` properly. Previously, if the `url_prefix` was `/foo` and the path received from a client was `/foo`, we would set *both* `SCRIPT_NAME` and `PATH_INFO` to `/foo`. This was incorrect. Now in such a case we set `PATH_INFO` to the empty string and we set `SCRIPT_NAME` to `/foo`. Note that the change we made has no effect on paths that do not match the `url_prefix` exactly (such as `/foo/bar`); these continue to operate as they did. See https://github.com/Pylons/waitress/issues/46
- Preserve header ordering of headers with the same name as per RFC 2616. See https://github.com/Pylons/waitress/pull/44
- When waitress receives a `Transfer-Encoding: chunked` request, we no longer send the `TRANSFER_ENCODING` nor the `HTTP_TRANSFER_ENCODING` value to the application in the environment. Instead, we pop this header. Since we cope with chunked requests by buffering the data in the server, we also know when a chunked request has ended, and therefore we know the content length. We set the content-length header in the environment, such that applications effectively never know the original request was a T-E: chunked request; it will appear to them as if the request is a non-chunked request with an accurate content-length.
- Cope with the fact that the `Transfer-Encoding` value is case-insensitive.
- When the `--unix-socket-perms` option was used as an argument to `waitress-serve`, a `TypeError` would be raised. See https://github.com/Pylons/waitress/issues/50.

CHAPTER 16

0.8.7 (2013-08-29)

- The HTTP version of the response returned by waitress when it catches an exception will now match the HTTP request version.
- Fix: CONNECTION header will be HTTP_CONNECTION and not CONNECTION_TYPE (see https://github.com/Pylons/waitress/issues/13)

CHAPTER 17

0.8.6 (2013-08-12)

- Do alternate type of checking for UNIX socket support, instead of checking for platform == windows.
- Functional tests now use multiprocessing module instead of subprocess module, speeding up test suite and making concurrent execution more reliable.
- Runner now appends the current working directory to `sys.path` to support running WSGI applications from a directory (i.e., not installed in a virtualenv).
- Add a `url_prefix` adjustment setting. You can use it by passing `script_name='/foo'` to `waitress.serve` or you can use it in a `PasteDeploy` ini file as `script_name = /foo`. This will cause the WSGI `SCRIPT_NAME` value to be the value passed minus any trailing slashes you add, and it will cause the `PATH_INFO` of any request which is prefixed with this value to be stripped of the prefix. You can use this instead of PasteDeploy's `prefixmiddleware` to always prefix the path.

CHAPTER 18

0.8.5 (2013-05-27)

- Fix runner multisegment imports in some Python 2 revisions (see https://github.com/Pylons/waitress/pull/34).
- For compatibility, WSGIServer is now an alias of TcpWSGIServer. The signature of BaseWSGIServer is now compatible with WSGIServer pre-0.8.4.

CHAPTER 19

0.8.4 (2013-05-24)

- Add a command-line runner called `waitress-serve` to allow Waitress to run WSGI applications without any addional machinery. This is essentially a thin wrapper around the `waitress.serve()` function.
- Allow parallel testing (e.g., under `detox` or `nosetests --processes`) using PID-dependent port / socket for functest servers.
- Fix integer overflow errors on large buffers. Thanks to Marcin Kuzminski for the patch. See: https://github.com/Pylons/waitress/issues/22
- Add support for listening on Unix domain sockets.

CHAPTER 20

0.8.3 (2013-04-28)

20.1 Features

- Add an `asyncore_loop_timeout` adjustment value, which controls the `timeout` value passed to `asyncore.loop`; defaults to 1.

20.2 Bug Fixes

- The default asyncore loop timeout is now 1 second. This prevents slow shutdown on Windows. See https://github.com/Pylons/waitress/issues/6 . This shouldn't matter to anyone in particular, but it can be changed via the `asyncore_loop_timeout` adjustment (it used to previously default to 30 seconds).
- Don't complain if there's a response to a HEAD request that contains a Content-Length > 0. See https://github.com/Pylons/waitress/pull/7.
- Fix bug in HTTP Expect/Continue support. See https://github.com/Pylons/waitress/issues/9 .

CHAPTER 21

0.8.2 (2012-11-14)

21.1 Bug Fixes

- https://corte.si/posts/code/pathod/pythonservers/index.html pointed out that sending a bad header resulted in an exception leading to a 500 response instead of the more proper 400 response without an exception.
- Fix a race condition in the test suite.
- Allow "ident" to be used as a keyword to `serve()` as per docs.
- Add py33 to tox.ini.

CHAPTER 22

0.8.1 (2012-02-13)

22.1 Bug Fixes

- A brown-bag bug prevented request concurrency. A slow request would block subsequent the responses of subsequent requests until the slow request's response was fully generated. This was due to a "task lock" being declared as a class attribute rather than as an instance attribute on HTTPChannel. Also took the opportunity to move another lock named "outbuf lock" to the channel instance rather than the class. See https://github.com/Pylons/waitress/pull/1 .

CHAPTER 23

0.8 (2012-01-31)

23.1 Features

- Support the WSGI `wsgi.file_wrapper` protocol as per https://www.python.org/dev/peps/pep-0333/#optional-platform-specific-file-handling. Here's a usage example:

```
import os

here = os.path.dirname(os.path.abspath(__file__))

def myapp(environ, start_response):
    f = open(os.path.join(here, 'myphoto.jpg'), 'rb')
    headers = [('Content-Type', 'image/jpeg')]
    start_response(
        '200 OK',
        headers
        )
    return environ['wsgi.file_wrapper'](f, 32768)
```

  The signature of the file wrapper constructor is `(filelike_object, block_size)`. Both arguments must be passed as positional (not keyword) arguments. The result of creating a file wrapper should be **returned** as the `app_iter` from a WSGI application.

  The object passed as `filelike_object` to the wrapper must be a file-like object which supports *at least* the `read()` method, and the `read()` method must support an optional size hint argument. It *should* support the `seek()` and `tell()` methods. If it does not, normal iteration over the filelike object using the provided block_size is used (and copying is done, negating any benefit of the file wrapper). It *should* support a `close()` method.

  The specified `block_size` argument to the file wrapper constructor will be used only when the `filelike_object` doesn't support `seek` and/or `tell` methods. Waitress needs to use normal iteration to serve the file in this degenerate case (as per the WSGI spec), and this block size will be used as the iteration chunk size. The `block_size` argument is optional; if it is not passed, a default value‘‘32768‘‘ is used.

Waitress will set a `Content-Length` header on the behalf of an application when a file wrapper with a sufficiently filelike object is used if the application hasn't already set one.

The machinery which handles a file wrapper currently doesn't do anything particularly special using fancy system calls (it doesn't use `sendfile` for example); using it currently just prevents the system from needing to copy data to a temporary buffer in order to send it to the client. No copying of data is done when a WSGI app returns a file wrapper that wraps a sufficiently filelike object. It may do something fancier in the future.

CHAPTER 24

0.7 (2012-01-11)

24.1 Features

- Default `send_bytes` value is now 18000 instead of 9000. The larger default value prevents asyncore from needing to execute select so many times to serve large files, speeding up file serving by about 15%-20% or so. This is probably only an optimization for LAN communications, and could slow things down across a WAN (due to higher TCP overhead), but we're likely to be behind a reverse proxy on a LAN anyway if in production.
- Added an (undocumented) profiling feature to the `serve()` command.

CHAPTER 25

0.6.1 (2012-01-08)

25.1 Bug Fixes

- Remove performance-sapping call to `pull_trigger` in the channel's `write_soon` method added mistakenly in 0.6.

CHAPTER 26

0.6 (2012-01-07)

26.1 Bug Fixes

- A logic error prevented the internal outbuf buffer of a channel from being flushed when the client could not accept the entire contents of the output buffer in a single succession of socket.send calls when the channel was in a "pending close" state. The socket in such a case would be closed prematurely, sometimes resulting in partially delivered content. This was discovered by a user using waitress behind an Nginx reverse proxy, which apparently is not always ready to receive data. The symptom was that he received "half" of a large CSS file (110K) while serving content via waitress behind the proxy.

CHAPTER 27

0.5 (2012-01-03)

27.1 Bug Fixes

- Fix PATH_INFO encoding/decoding on Python 3 (as per PEP 3333, tunnel bytes-in-unicode-as-latin-1-after-unquoting).

CHAPTER 28

0.4 (2012-01-02)

28.1 Features

- Added "design" document to docs.

28.2 Bug Fixes

- Set default `connection_limit` back to 100 for benefit of maximal platform compatibility.
- Normalize setting of `last_activity` during send.
- Minor resource cleanups during tests.
- Channel timeout cleanup was broken.

CHAPTER 29

0.3 (2012-01-02)

29.1 Features

- Dont hang a thread up trying to send data to slow clients.
- Use self.logger to log socket errors instead of self.log_info (normalize).
- Remove pointless handle_error method from channel.
- Queue requests instead of tasks in a channel.

29.2 Bug Fixes

- Expect: 100-continue responses were broken.

CHAPTER 30

0.2 (2011-12-31)

30.1 Bug Fixes

- Set up logging by calling logging.basicConfig() when `serve` is called (show tracebacks and other warnings to console by default).
- Disallow WSGI applications to set "hop-by-hop" headers (Connection, Transfer-Encoding, etc).
- Don't treat 304 status responses specially in HTTP/1.1 mode.
- Remove out of date `interfaces.py` file.
- Normalize logging (all output is now sent to the `waitress` logger rather than in degenerate cases some output being sent directly to stderr).

30.2 Features

- Support HTTP/1.1 `Transfer-Encoding: chunked` responses.
- Slightly better docs about logging.

CHAPTER 31

0.1 (2011-12-30)

- Initial release.

CHAPTER 32

Known Issues

- Does not support TLS natively. See *Using Behind a Reverse Proxy* for more information.

CHAPTER 33

Support and Development

The Pylons Project web site is the main online source of Waitress support and development information.

To report bugs, use the issue tracker.

If you've got questions that aren't answered by this documentation, contact the Pylons-discuss maillist or join the #pyramid IRC channel.

Browse and check out tagged and trunk versions of Waitress via the Waitress GitHub repository. To check out the trunk via `git`, use this command:

```
git clone git@github.com:Pylons/waitress.git
```

To find out how to become a contributor to Waitress, please see the guidelines in contributing.md and How to Contribute Source Code and Documentation.

CHAPTER 34

Why?

At the time of the release of Waitress, there are already many pure-Python WSGI servers. Why would we need another?

Waitress is meant to be useful to web framework authors who require broad platform support. It's neither the fastest nor the fanciest WSGI server available but using it helps eliminate the N-by-M documentation burden (e.g. production vs. deployment, Windows vs. Unix, Python 3 vs. Python 2, PyPy vs. CPython) and resulting user confusion imposed by spotty platform support of the current (2012-ish) crop of WSGI servers. For example, `gunicorn` is great, but doesn't run on Windows. `paste.httpserver` is perfectly serviceable, but doesn't run under Python 3 and has no dedicated tests suite that would allow someone who did a Python 3 port to know it worked after a port was completed. `wsgiref` works fine under most any Python, but it's a little slow and it's not recommended for production use as it's single-threaded and has not been audited for security issues.

At the time of this writing, some existing WSGI servers already claim wide platform support and have serviceable test suites. The CherryPy WSGI server, for example, targets Python 2 and Python 3 and it can run on UNIX or Windows. However, it is not distributed separately from its eponymous web framework, and requiring a non-CherryPy web framework to depend on the CherryPy web framework distribution simply for its server component is awkward. The test suite of the CherryPy server also depends on the CherryPy web framework, so even if we forked its server component into a separate distribution, we would have still needed to backfill for all of its tests. The CherryPy team has started work on Cheroot, which should solve this problem, however.

Waitress is a fork of the WSGI-related components which existed in `zope.server`. `zope.server` had passable framework-independent test coverage out of the box, and a good bit more coverage was added during the fork. `zope.server` has existed in one form or another since about 2001, and has seen production usage since then, so Waitress is not exactly "another" server, it's more a repackaging of an old one that was already known to work fairly well.

Python Module Index

W

Index