Paste

Release 1.7.5.1

November 16, 2016

1	News	3
2	The Future Of Paste	21
3	Testing Applications with Paste	23
4	URL Parsing With WSGI And Paste	27
5	A Do-It-Yourself Framework	33
6	The Paste HTTP Server Thread Pool	43
7	Features	47
8	Python Paste Developer Guide	51
9	Paste Style Guide	53
10	Community	55
11	Mailing Lists	57
12	Repository	59
13	Downloads	61
14	Indices and tables	63
15	Other Components	65
16	License	67

Contents:

CHAPTER 1

News

Contents

- *News*
  - *2.0.3*
  - *2.0.2*
  - *2.0.1*
  - *2.0*
  - *1.7.5.1*
  - *1.7.5*
  - *1.7.4*
  - *1.7.3.1*
  - *1.7.3*
  - *1.7.2*
  - *1.7.1*
  - *1.7*
  - *1.6.1*
  - *1.6*
  - *1.5.1*
  - *1.5*
  - *1.4.2*
  - *1.4.1*
  - *1.4*
  - *1.3*
  - *1.2.1*
  - *1.2*
    - *In paste.wsgiwrappers*
    - *In paste.httpserver*
  - *1.1.1*
  - *1.1*
  - *1.0*
  - *0.9.9*
  - *0.9.8*
  - *0.9.7*
  - *0.9.6*
  - *0.9.5*
  - *0.9.4*
  - *0.9.3*
  - *0.9.2*
  - *0.9.1*
  - *0.9*
  - *0.5*
  - *0.4.1*
  - *0.4*

1.1 2.0.3

- #26: Change six requirement to >=1.4.0 from [Linus Heckemann](https://bitbucket.org/sphalerite/) https://bitbucket.org/ianb/paste/pull-requests/26/change-six-requirement-to-140/diff
- #28: Py3k fixes from [Nils Philippsen](https://bitbucket.org/nilsph/) https://bitbucket.org/ianb/paste/pull-requests/28/py3k-fixes/diff
- #29: paste.wsgilib.add_close: Add __next__ method to support using *add_close* objects as itera-

tors on Python 3. fixes https://bitbucket.org/ianb/pastedeploy/issues/18/py3-test_config_middleware-failed from [Marc Abramowitz](https://bitbucket.org/msabramo/) https://bitbucket.org/ianb/paste/pull-requests/29/pastewsgilibadd_close-add-__next__-method/diff

- #30: tox.ini: Add py35 to envlist from [Marc Abramowitz](https://bitbucket.org/msabramo/) https://bitbucket.org/ianb/paste/pull-requests/30/toxini-add-py35-to-envlist/diff
- #31: Enable testing with pypy from [Marc Abramowitz](https://bitbucket.org/msabramo/) https://bitbucket.org/ianb/paste/pull-requests/31/enable-testing-with-pypy/diff
- #33: tox.ini: Measure test coveraage from [Marc Abramowitz](https://bitbucket.org/msabramo/) https://bitbucket.org/ianb/paste/pull-requests/33/toxini-measure-test-coverage/diff

1.2 2.0.2

- #22: Fix improper commas in request headers in wsgi_environ (https://bitbucket.org/ianb/paste/pull-request/22/fix-improper-commas-in-request-headers-in) Fixes issue #4 ("WSGI environ totally borked") (https://bitbucket.org/ianb/paste/issue/4/wsgi-environ-totally-borked)
- #24: test_wsgirequest_charset: Use UTF-8 instead of iso-8859-1 (https://bitbucket.org/ianb/paste/pull-request/24/test_wsgirequest_charset-use-utf-8-instead) Fixes issue #7 ("Python 3 test failure") (https://bitbucket.org/ianb/paste/issue/7/python-3-test-failure)
- #23: Replace cgi.parse_qsl w/ six.moves.urllib.parse.parse_qsl (https://bitbucket.org/ianb/paste/pull-request/23/replace-cgiparse_qsl-w) Fixes issue #8 ("cgi.parse_qsl is pending deprecation") (https://bitbucket.org/ianb/paste/issue/8/cgiparse_qsl-is-pending-deprecation)
- #20: Escape CGI environment variables in HTTP 404 responses (https://bitbucket.org/ianb/paste/pull-request/20/escape-cgi-environment-variables-in-http)
- #6: Add HTTP exception for new code 429 "Too Many Requests" (https://bitbucket.org/ianb/paste/pull-request/6/add-http-exception-for-new-code-429-too)
- #25: replace `has_key` method to `in` operator #9 (https://bitbucket.org/ianb/paste/pull-request/25/replace-has_key-method-to-in-operator-9) Fixes #9 ("used methods removed from py3") (https://bitbucket.org/ianb/paste/issue/9/used-methods-removed-from-py3)
- #5: Invalid error message when the socket is already in use (https://bitbucket.org/ianb/paste/issue/5/invalid-error-message-when-the-socket-is)

1.3 2.0.1

- Fix setup.py for six dependency: move the six dependency from extras_require to install_requires.
- Port paste.proxy to Python 3.
- Fix paste.exceptions.serial_number_generator.hash_identifier() on Python 3.
- Fix paste.util.threadedprint.uninstall(). Rename duplicated uninstall() function to uninstall_stdin() and fix typo in variable name (_oldstin => _oldstdin).
- Add README.rst file.

1.4 2.0

- Experimental Python 3 support.

- paste now requires the six module.
- Drop support of Python 2.5 and older.
- Fixed `egg:Paste#cgi`
- In `paste.httpserver`: give a 100 Continue response even when the server has been configured as an HTTP/1.0 server (clients may send `Expect: 100-Continue` before they know the version), and wrap 100 Continue `environ['wsgi.input']` files with LimitedLengthFile just like normal request bodies are wrapped, keeping WSGI applications from over-reading from the socket.
- Fixed parsing of paths beginning with multiple forward slashes.
- Add tox.ini to run tests with tox on Python 2.6, 2.7 and 3.4.

1.5 1.7.5.1

- Fix bug introduced in `paste.auth.auth_tkt` (with `url_unquote`)

1.6 1.7.5

- Won't install `tests/` directory (also caused installation problems on some Mac systems).
- Fixed problem with gzip middleware and zero-length responses.
- Use `X-Forwarded-For` header in `paste.translogger`
- Fixed problems with mimeparse code
- Fixed some corner cases with CGI scripts
- `paste.auth.auth_tkt` will URL-quote usernames, avoiding some errors with usernames with `!` in them.
- Improve handling of errors in fetching error pages in `paste.errordocument`.

1.7 1.7.4

- Fix XSS bug (security issue) with not found handlers for `paste.urlparser.StaticURLParser` and `paste.urlmap.URLMap`. If you ask for a path with `/--><script>...` that will be inserted in the error page and can execute Javascript. Reported by Tim Wintle with further details from Georg-Christian Pranschke.
- Replaced `paste.util.mimeparse.desired_match()`

1.8 1.7.3.1

- Removed directory name from 404 errors in `paste.urlparser.StaticURLParser`.
- Fixed packaging to include Javascript and images for `paste.evalexception`

1.9 1.7.3

- I got a fever and the only prescription is more `paste.cowbell`!
- Fix `paste.httpserver` on Python 2.6.
- Fix `paste.auth.cookie`, which would insert newlines for long cookies.
- `paste.util.mimeparse` parses a single `*` in Accept headers (sent by IE 6).
- Fix some problems with the `wdg_validate` middleware.
- Improvements to `paste.auth.auth_tkt`: add httponly support, don't always aggressively set cookies without the `wildcard_cookie` option. Also on logout, make cookies expire.
- In `paste.proxy.Proxy` handle Content-Length of -1.
- In `paste.httpexceptions` avoid some unicode errors.
- In `paste.httpserver` handle `.read()` from 100 Continue properly (because of a typo it was doing a readline).
- Update `paste.util.mimeparse` from upstream.

1.10 1.7.2

- In `paste.proxy`, added some more headers that are disallowed in WSGI (e.g., Keep-Alive). Send Content-Length. Also fix the missing query string when using `paste.proxy.Proxy` (`paste.proxy.TransparentProxy` already worked).
- Make `paste.debug.prints` work with Google App Engine.
- Make `environ['wsgi.input']` with `paste.httpserver` only have a `seek` method if it is wrapping something with a seek method (which usually it is not).
- In `paste.httpserver` re-raise KeyboardInterrupt in worker threads.
- Added support for the `HttpOnly` Cookie property to `paste.wsgiwrappers`
- Added `paste.reloader.add_file_callback()`, which lets you watch files based on a callback.
- Quiet Python 2.6 deprecation warnings.
- Fix `paste.auth.cookie` generating bad headers.
- Added `paste.reloader.JythonMonitor` for an experimental, optimized reloader on Jython.

1.11 1.7.1

- Normalize and make absolute the paths passed to `paste.urlparser.StaticURLParser` (before passing a relative-to-cwd path to that class would cause Forbidden errors).
- Deprecate `paste.fixture.setup_module()`

1.12 1.7

- Fixed bug in `paste.fixture.TestApp` that would submit forms with unnamed fields (like an unnamed submit button). Also made checkboxes with no explicit `value` send `on` instead of `checked` (which is what browsers do).
- Fixed bug in `paste.httpserver` where `environ['wsgi.input'].readline(max_size)` ignored the max_size argument, which can lead to large memory usage (from Jakub Stolarski)
- Make `paste.cascade` notice sockets that have stopped producing data. From Casey Zednick.
- In `paste.fixture.TestApp` Accept MultiDict values for the `params` argument in requests. (Anything with a `.items()` method will have its items encoded as the request parameters.)
- Fix `paste.httpserver` to allow binding to port 0.
- In `paste.auth.auth_tkt`, set the same cookies (with the same domains) in `set_cookie` as get unset in `logout_user_cookie`.
- In `paste.translogger` save REQUEST_METHOD in case it gets overridden somewhere (e.g., when using errordocuments POST would show up as GET).
- Exceptions with unicode messages don't cause the collector to fail.
- Sometimes `paste.exceptions.errormiddleware.ErrorMiddleware` would not call start_response properly; this is fixed (from Andreas Kloecker).
- `paste.fixture.TestApp` can store multiple cookie values (previously only one cookie was stored; from Andrey Lebedev)
- `u'' in TestApp(app).get('/')` will work when the body isn't ASCII (before it would give a unicode error). This problem wasn't present in the recommended WebTest.
- `paste.debug.profile` won't break when content is served with no Content-Type.
- Accept relative paths and paths with `/../` in them for `paste.urlparser.StaticURLParser` (from Eric Larson). Also fix problem with case normalization on Windows (from Ionel Maries Cristian).
- `paste.registry.StackedObjectProxy`'s now include the proxied object's names via `__dir__` (for Python 2.6).
- Use `environ['wsgi.file_wrapper']` when available (in `paste.fileapp`).
- Make `paste.debug.prints` compatible with App Engine.
- Fix the `domain` keyword in `paste.wsgiwrappers.WSGIResponse.delete_cookie()`.

1.13 1.6.1

- Fixed bug in paste lint where PATH_INFO would become unicode.

1.14 1.6

- Make the import of `socket.sslerror` conditional in `paste.exceptions.reporter` (needed for Python interpreters compiled without SSL support).
- In `paste.proxy.TransparentProxy`, don't overwrite `X-Forwarded-For` header if it is already in the environment.

- Added `226 IM Used` status code to `paste.wsgiwrappers`
- In `paste.fixture.TestApp` treat `<image type="image">` the same as a submit button.
- Use `OpenSSL.tsafe.Connection` for https with `paste.httpserver`, avoiding some possible errors (`sslv3 alert bad record mac`).
- Fix small issue with `paste.cgiapp` and mod_wsgi.
- Use `BaseCookie` instead of `SimpleCookie` for storing cookies (avoids quoting cookie values).

1.15 1.5.1

- Make `paste.cascade` more tolerant of a missing or invalid Content-Length.

1.16 1.5

- Fixed memory leak with `paste.registry` not properly removing all references to registered objects should register be called multiple times during a single context for a StackedObjectProxy.
- `paste.httpheaders.CONTENT_RANGE` returns `bytes START-END/LENGTH` instead of just `START-END/LENGTH`
- In `paste.fixture.TestApp` set `CONTENT_TYPE` to `'application/x-www-form-urlencoded'` whenever there are parameters (and no other content type was provided).
- In `paste.session`, when cleaning files ignore files that aren't session files.
- `paste.httpexceptions.HTTPExceptionHandler` will no longer catch exceptions raised during the app_iter iteration.
- `paste.cascade.Cascade` copies `wsgi.input` when cascading, so that one of the applications cannot read the input and leave a later application blocked when it tries to read the input.
- Fix assigning to `WSGIResponse.charset` breaking the content-type.
- SMTP authentication is supported for the exception handler. You may now set `smtp_username`, `smtp_password` and `smtp_use_tls` to control this behavior. From pthy.

1.17 1.4.2

- Remove FreeBSD threadpool condition in paste.httpserver (which was also breaking code for Windows users).
- Fix problem with `paste.wsgilib.intercept_output` and passing up exceptions.

1.18 1.4.1

- Allow customization of the `paste.config.ConfigMiddleware` environ key.
- Added a `current` method (an alias of `current_conf`) to `paste.config.DispatchingConfig`.
- Make test response `.form` attribute work when you have a single named form.
- Try to encode any unicode input to `paste.auth.auth_tkt`

- `paste.wsgiwrappers.WSGIResponse` now has a `.content_type` attribute (that does not include parameters), and a `.charset` attribute (that gets the charset parameter).
- Inherit inherit show_exceptions_in_wsgi_errors from global configuration. Inherit `debug` more properly.

1.19 1.4

- In `paste.httpserver` added lots of functionality to the threadpool. See the paste.httpserver threadpool documentation for details. This catches worker threads (and WSGI apps) that take too long to complete their task; killing them eventually, adding more worker threads when the pool is exhausted and it doesn't look good that it'll clear soon, and optionally killing the process when there are too many lost/zombie threads (you must be using some kind supervisor process for this last response to make sense).
- Save host and scheme information during real HTTP proxy requests to `paste.httpserver`, into the keys `paste.httpserver.proxy.scheme` and `paste.httpserver.proxy.host`
- In `paste.exceptions` always call `start_response`; may help problems when there is an exception in `start_response` itself.
- Added method to `paste.registry.StackedObjectProxy`, `_object_stack()`, which returns a list of all the registered objects. Useful if you want to search through the effective history of a stacked object.
- Fixed infinite recursion problem with `paste.request.EnvironHeaders.keys()`.
- Fix `paste.wsgiwrappers.WSGIRequest.urlvars` to use `wsgiorg.routing_args`
- Remove port from `paste.request.construct_url` if it's the default port (e.g., port 80 for `http`).
- `paste.proxy` works with headers with continuations in the response (i.e., a header that spans multiple lines). Also, treat a missing Content-Length as 0, not unlimited (may have previously caused freeze ups for some kinds of requests).
- `StackedObjectProxy` supports `__call__` (i.e., you can use `StackedObjectProxy` with callable objects).
- Fixed `ProfileMiddleware` not calling `close()` on consumed app_iters.
- `httpheaders.AcceptLanguage` now won't give an exception when there is a malformed parameter in the header.
- Fix `paste.auth.form.auth_form` Paste Deploy entry point.
- Added REST methods to `paste.fixture.TestApp`, so you can more easily do requests like PUT and DELETE. From Anders Pearson.
- Added `{{default var=default_value}}` command to `paste.util.template`. Make `{{# comment}}` work.

1.20 1.3

- In `paste.httpserver` remove the reverse DNS lookup to set `REMOTE_HOST`
- In `paste.fileapp`, if the client sends both If-None-Match and If-Modified-Since, prefer If-None-Match. Make ETags include the size as well as last modified timestamp. Make it possible to override how mimetypes are guessed.
- `HTTPException` objects now have a `exc.response(environ)` method that returns a `WSGIResponse` object.

- `egg:Paste#watch_threads` will show tracebacks of each thread under Python 2.5.
- Made `paste.util.template` trim whitespace around statements that are on their own line.
- `paste.fileapp.DataApp` now accepts `allowed_headers=[...]` to specify the allowed headers. By default only `GET` and `HEAD` are allowed.
- Added `paste.util.import_string.try_import_module`, which imports modules and catches `ImportError`, but only if it's an error importing the specific module, not an uncaught `ImportError` in the module being imported.

1.21 1.2.1

- `paste.httpserver` didn't implement the `readline` that the `cgi` module wants (regression in 1.2).

1.22 1.2

- **Backward incompatible change**: `paste.fileapp.FileApp` properly supports request methods, including HEAD. If you were subclassing `FileApp` or `DataApp` and overriding `__call__()` you may have to subclass `get()` instead.
- paste.httpheaders now parses the HTTP Accept-Language header and returns a list of languages the browser supports in the order it prefers them.
- paste.mimeparse module added that handles parsing HTTP Accept headers for quality and mime-types.
- `paste.request.construct_url` was adding `SERVER_PORT` to `HTTP_HOST`; but `HTTP_HOST` (from the Host header) generally contains a port when necessary, and `SERVER_PORT` should only be used with `SERVER_NAME`.
- Added entry point for `paste.registry.RegistryManager` (`egg:Paste#registry`).
- `paste.request.HeaderDict` fixed to know that `Content-Length` maps to `CONTENT_LENGTH`.
- Can use `paste.urlparser.StaticURLParser` with sub-instances other than `paste.fileapp.FileApp` (if you subclass and override `make_app`)
- `paste.fixture.TestApp.get(status=X)` takes a list of allowed status codes for `X`.
- Added a small templating system for internal use (`paste.util.template`)
- Removed a bunch of long-deprecated modules (generally modules that have been moved to other names).

1.22.1 In paste.wsgiwrappers

- `paste.wsgiwrappers.WSGIRequest` has match_accept() function to screen incoming HTPT Accept values against a list of mime-types.
- `paste.wsgiwrappers.WSGIRequest.defaults` now accepts a new key:

  `language:` The i18n language that should be used as the fallback should a translation not occur in a language file. See docs for details.
- `paste.wsgiwrappers.WSGIRequest` can now optionally decode form parameters to unicode when it has a `charset` value set.
- Deprecated the `paste.wsgiwrappers.settings` StackedObjectProxy dictionary for `paste.wsgiwrappers.WSGIResponse.defaults`.

1.22.2 In paste.httpserver

- Regression in 1.1 fixed, where Paste's HTTP server would drop trailing slashes from paths.
- `paste.httpserver` now puts a key in the environment when using a thread pool that allows you to track the thread pool and see any wedged threads. `egg:Paste#watch_threads` is an application that can display this information.
- `paste.httpserver` now accepts all request methods, not just `GET`, `PUT`, etc. (Methods like `MKCOL` were previously rejected.)
- `paste.httpserver` has a `wsgi.input` that now does not block if you try to read past the end (it is limited to returning the number of bytes given in `Content-Length`). Double-reading from `wsgi.input` won't give you the same data, but it won't cause blocking.

1.23 1.1.1

- Fixed major issue with serving static files on Windows (a regression in Paste 1.1 where most static files would return 404 Not Found).
- Fixed `parse_dict_querystring` returning empty dicts instead of `MultiDict`s.
- Added `paste.config`, a rewrite of `paste.deploy.config` using `paste.registry`. This version of `ConfigMiddleware` will enable use of `paste.config.CONFIG` within the `EvalException` interactive debugger.
- Fixed problem where `paste.recursive` would leave `wsgi.input` and `CONTENT_LENGTH` set for recursive requests.
- Changed the static file servers to give 404 Not Found responses when you have extra parts after a static file, instead of 400 Bad Request (like when you request `/file.html/extra/path`)

1.24 1.1

- Security fix for `paste.urlparser.StaticURLParser`. The problem allowed escaping the root (and reading files) when used with `paste.httpserver` (this does not effect other servers, and does not apply when proxying requests from Apache to `paste.httpserver`).
- `paste.httpserver` and `paste.fixture.TestApp` url-unquote `SCRIPT_NAME` and `PATH_INFO`, as specified in the CGI spec. Thanks to Jon Nelson for pointing out both these issues.
- `paste.registry` now works within the `EvalException` interactive debugger.
- Fixed `paste.auth.open_id` failures not returning a correct response.
- Changed `paste.httpexceptions.HTTPUnauthorized` so that the `WWW-Authenticate` header is not required. 401 responses don't *have* to have that header.
- In `paste.fixture.TestApp`: `<form>` tags that have to `action` will preserve the existing query string. (Generally relative links that are completely empty should but were not preserving the query string)
- Made `paste.*` compatible with py2exe by adding a `modulefinder` call in `__init__.py`
- The `paste.gzipper` gzipping middleware wasn't changing the Content-Length header properly; thanks to Brad Clements for the fix.

- Fixed `paste.proxy` to not use anything based on the dict form of `httplib..HTTPMessage`. This form folds headers together in a way that breaks `Set-Cookie` headers (two `Set-Cookie` headers would be merged into one).
- `paste.request.parse_formvars` didn't accept parameters in `CONTENT_TYPE`. `prototype.js` sets a `charset` parameter, which caused a problem.
- Added a `__traceback_decorator__` magic local variable, to allow arbitrary manipulation of the output of `paste.exceptions.collector` before formatting.
- Added unicorn power to `paste.pony` (from Chad Whitacre)
- For `paste.httpserver` SSL support: add support loading an explicit certificate context, and using `ssl_pem='*'` create an unsigned SSL certificate (from Jason Kirtland).
- Fix some cases where `paste.httpserver` can have an orphaned thread pool (causing the process to not shut down properly). Patch from jek.

1.25 1.0

- Fixed `parsed_formvars` potentially locking up on wsgi.input after modification of `QUERY_STRING`.
- Fixed problem where `paste.exceptions.errormiddleware` didn't expose the `.close()` method of the app_iter that it wraps (to catch exceptions). This is a problem if the server about the errormiddleware aborts the request; it should then call `.close()`, but won't necessarily exhaust the iterator to do so.
- Added entry point for `paste.translogger` (`egg:Paste#translogger`)
- Fixed some cases where long data (e.g., a file upload) would show up in the error report, creating a very very large report. Also, put in a monkeypatch for the `cgi` module so that `repr(uploaded_field)` won't load the entire field into memory (from its temporary file location).
- Added a `force_host` option to `paste.proxy.TransparentProxy`, which will force all incoming requests to the same host, but leave the `Host` header intact.
- Added automatic cleanup of old sessions for `paste.session`, from Amir Salihefendic.
- Quote the function name in tracebacks in the exception formatter; Genshi has function names that use <>.

1.26 0.9.9

- Fixed `paste.response.HeaderDict` `get` and `setdefault` methods to be case insensitive
- Fix use of `TestApp().post(params={'key': ['list of', 'values']})` as reported by Syver Enstad.
- `paste.fileapp.DataApp` is now directly usable (was previously only usable as an abstract base class).

1.27 0.9.8

- Fixed `wsgiwrappers.WSGIResponse.delete_cookie`. It also now takes optional path and domain arguments
- `wsgiwrappers.WSGIResponse` now handles generator/iterator content more cleanly, and properly encodes unicode content according to its specified charset

- Fixed `wsgiwrappers.WSGIResponse` mishandling multiple headers of the same name
- Added a Paste Deploy entry point for `paste.auth.cookie`
- Moved Paste Deploy dependencies out of top-level modules and into Paste-Deploy-specific entry point functions. This should make Paste more-or-less Paste Deploy independent. `paste.urlparser` and `paste.exceptions.errormiddleware` still have some leftover bits.
- Added another redirector type to `paste.recursive`, `environ['paste.recursive.include_app_iter']` which gives access to the original app_iter.
- Bug with `wsgilib.catch_errors` and app_iters with no `close()` method.
- Long words in tracebacks weren't being wrapped correctly at all. Also, large data would cause the wrapping routine to give a recursion error. Now large data is truncated (at 1000 characters), and recursion won't be a problem. Also, wrapping shouldn't lose characters.
- Better exception if you try to put a non-str into environ when using `paste.auth.cookie`
- `paste.exceptions.collector` produces an `exc_data.exception_type` that is a class, not a string. This helps it get formatted better in Python 2.5.
- All the tests pass on Python 2.5!
- Added `paste.proxy.TransparentProxy`, which just sends the request described in the WSGI environ on without any modification. More useful for WSGI clients than servers, it effectively allows any WSGI-based request mechanism to act like an httplib-based request mechanism.
- Added a `cache_max_age` argument to `paste.urlparser.StaticURLParser`, which allows you to encourage the caching of static files. Patch from Brad Clements.
- Added `suppress_http_headers` to `paste.proxy.Proxy`, which will filter out HTTP headers from the request before passing it on. Patch from Brad Clements.

1.28 0.9.7

- The `EvalException` 'full traceback' button is now only displayed when the full traceback differs from the regular (includes hidden frames).
- Fixed `EvalException` returning a Content-type of 'text-html' instead of 'text/html' in some cases.

1.29 0.9.6

- Renamed the `paste.util.multidict.multidict` class to `paste.util.multidict.MultiDict`

1.30 0.9.5

- Fixed a security vulnerability in `paste.urlparser`'s StaticURLParser and PkgResourcesParser where, with some servers, you could escape the document root.
- Significantly improved `paste.httpserver`'s (egg:Paste#http) performance. It now uses a thread pool: previously it created a new thread for every request. To revert back to the old, slower behavior, set:

```
use_threadpool = false
```

  in the [server:main] section of the config file.

- More control of where the output of `paste.debug.prints` goes.
- Added a warning to `paste.wsgilib.add_close` if the upstream app_iter consumer doesn't call the `app_iter.close()` method.
- Fixed `testapp.post(params={})`
- Fixed `paste.translogger.TransLogger` to log to the Apache combined log format as advertised.
- Fixed `paste.urlparser` classes to handle quoted characters (e.g. %20) in URL paths.
- Changed `paste.session` to allow manipulating a session for the first time after `start_response` is called.
- Added `paste.wsgilib.add_start_close` which calls a function just before returning the first chunk of the app_iter.
- Changed `paste.urlmap` so that it matches domain-specific mappings before domain-neutral mappings.
- Fixed IE 6 potentially receiving the following `"400 Bad Request"` error on file downloads:

```
Please check your system clock.
According to this server, the time provided in the
If-Modified-Since header is in the future.
```

- Added a 'no' keyword argument to `TestResponse.mustcontain`, so you can assert that a response does contain some strings at the same time that you assert that a response *does not* contain other strings, like:

```
res = app.get('/')
res.mustcontain('this must be there',
                no=['error', 'unexpected'])
```

- Fixed `fileapp.FileApp` to pay attention to the `If-None-Match` header, which does ETag matching; before only `If-Modified-Since` was supported, even though an `ETag` header was being sent; in particular Firefox would then only send `If-None-Match` and so conditional requests never worked.
- Changed usage of `paste.request.MultiDict` to `paste.util.multidict`, particularly in `paste.wsgiwrappers` where `request.GET` returns a new style of dictionary interface.
- Be more careful in `paste.request.parse_formvars` not to let the `cgi` module read from `wsgi.input` when there are no parsable variables in the input (based on `CONTENT_TYPE`).

1.31 0.9.4

- This released was lost in a tragic clerical accident.

1.32 0.9.3

- 0.9.2 Included a version of MochiKit that was no longer compatible with evalexception; 0.9.3 reverts to a previous version.
- Change wsgi.run_once=False for `paste.httpserver`
- Added entry points for debug apps

1.33 0.9.2

- Fix in paste.urlmap when connecting with host:port.
- Added `/_debug/summary` to evalexception, which gives a JSON-formatted list of all the exceptions in memory.

1.34 0.9.1

- A fix for paste.errordocument, when doing an internal redirect from a POST request (the request is rewritten as a GET request)

1.35 0.9

- Added paste.request.WSGIRequest, a request object that wraps the WSGI environment.
- Added paste.registry, which is middleware for registering threadlocal objects in a request.
- Avoid annoying warning from paste.recursive
- `paste.httpserver` now removes HTTPServer's transaction logging, which was doing a reverse DNS lookup.
- Added `has_session` to `paste.session`
- Allow for conditional `paste.wsgilib.intercept_output` which should be slightly faster (and streamable) compared to doing the condition manually.
- Added entry point for paste.proxy, plus improvements from Brad Clements (support path in target, filter request methods)
- Added paste.pony so pony power can be added to any WSGI application.
- Added port matching to `paste.urlmap`.

1.36 0.5

- Added paste.auth.auth_tkt
- Added paste.auth.grantip

1.37 0.4.1

- Some bug fixes to the built-in HTTP server.
- Experimental paste.progress middleware for tracking upload progress
- Some tweaking of how paste.reload works, especially with respect to shutdown.

1.38 0.4

- Fixed up paste documentation (especially for new packages/modules)
- Added paste.auth package for authentication related WSGI middle-ware components:
  - `basic` and `digest` HTTP authentication as described by RFC 2617
  - support for Yale's Central Authentication System (`cas`)
  - `open_id` supports single sign-on originally developed for LiveJournal (see http://openid.net)
  - `cookie` digitally signs cookies to record the current authenticated user (`REMOTE_USER`), session identifier (`REMOTE_SESSION`), and other WSGI entries in the `environ`.
  - a `form` module (to be used with `cookie` or an equivalent) provides a simple HTML based form authentication.
  - the `multi` module is an *experimental* mechanism for choosing an authentication mechanism based on the WSGI `environ`
- Added paste.httpserver module which provides a very simple WSGI server built upon python's `BaseHTTPServer`; this server has support for several features:
  - support for SSL connections via OpenSSL
  - support for HTTP/1.1 `100 Continue` messages as required by the WSGI specification (many HTTP server implementations don't do this)
  - implemented as a Mix-In so that it can be used with other more enchanted versions of `BaseHTTPServer`
  - support for 'Keep-Alive' (standard in HTTP/1.1) by either providing a content-length or closing a connection if one is not available
- Improved the paste.httpexceptions module:
  - added missing exception objects, and better descriptions
  - fixed several bugs in how exceptions are caught and propagated
  - usage as a `wsgi_application()` enables exceptions to be returned without throwing or catching the error
  - support for plain/text messages for text-only clients such as curl, python's urllib, or Microsoft Excel
  - allows customization of the HTML template for higher-level frameworks
- Added paste.httpheaders module to provide a uniform mechanism to access/update standard HTTP headers in a WSGI `environ` and `response_headers` collection; it includes specific support for:
  - providing "common" header names and sorting them as suggested by RFC 2616
  - validated support for `Cache-Control` header construction
  - validated support for `Content-Disposition` header construction
  - parsing of `If-Modified-Since` and other date oriented headers
  - parsing of Range header for partial-content delivery
  - composition of HTTP/1.1 digest `Authorization` responses
- Improved paste.fileapp to support:
  - static in-memory resources
  - incremental downloading of files from disk

  - responding to 'Range' requests to handle partial downloads
  - allowing cache settings to be easily provided; including support for HTTP/1.0 'Expires' and HTTP/1.1 'Cache-Control'
- Added an *experimental* paste.transaction module for handling commit/rollback of standard DBAPI database connections
- Added a paste.util.datetimeutil module for parsing standard date/time user-generated text values
- Added a debug package, which includes:
  - previous top-level modules `prints`, `profile`, `wdg_validate` and `doctest_webapp`
  - a `testserver` module suitable to test HTTP socket connections via `py.test`
- Re-factored paste.wsgilib into several other modules:
  - functions regarding header manipulation moved to paste.response
  - functions regarding cookies and arguments moved to paste.request
- Significant improvements to `wsgiutils.wsgilib` module:
  - added a `dump_environ` application to help debugging
  - fixes to `raw_interactive` to comply with WSGI specifications
  - `raw_interactive` now logs all 5xx exceptions and sets HTTP_HOST
- Added an argument `no_profile` to paste.debug.profile.profile_decorator; if that option is false, then don't profile the function at all.
- Changed paste.lint to check that the status contains a message (e.g., `"404 Not Found"` instead of just `"404"`). Check that environmental variables `HTTP_CONTENT_TYPE` and `HTTP_CONTENT_LENGTH` are no present. Made unknown `REQUEST_METHOD` a warning (not an error).
- Added parameter `cwd` to TestFileEnvironment.run
- paste.fixture.TestApp:
  - Form filling code (use `response.forms[0]` to get a form object)
  - Added click method.
  - Better attribute errors.
  - You can force set hidden fields using `form.fields[name].force_value(value)` (normally setting the value of a hidden field is an error).
  - Frameworks can now add custom attributes to the response object.
- `paste.wsgilib.capture_output` is deprecated in favor of paste.wsgilib.intercept_output
- Remove use of exceptions in paste.cascade.Cascade, which causes weird effects in some cases. Generally we aren't using exceptions internally now, only return status codes. Also in cascade, be careful to keep cascaded requests from sharing the same environment.
- `paste.wsgilib.error_response` is deprecated (paste.httpexceptions replaces this with exception's `.wsgi_application` method).
- Moved `paste.login` to the attic, since paste.auth pretty much replaces it.
- paste.urlparser improvements:
  - Added an application urlparser.StaticURLParser for serving static files.

- Added an application urlparser.PkgResourcesParser for serving static files found with `pkg_resources` (e.g., out of zipped Eggs).
- Be less picky about ambiguous filenames when using URLParser; if an exact file match exists, use that. (`file.gif.bak` would cause a request for `file.gif` to be ambiguous before)
- Now looks for a `.wsgi_application` attribute when serving Python files/modules, as a general hook for returning a WSGI application version of an object.

* The ErrorMiddleware:
  - Returns trimmed-down exceptions if there is a _ GET variable in the request (which is meant to signal an XMLHttpRequest). Exceptions displayed in this context are best when they are smaller and easier to display.
  - Includes a text version of the traceback, for easier copy-and-paste.
  - Avoid printing exceptions to `wsgi.errors` if they are already displayed elsewhere (at least by default).
  - Highlight Python code.
* Use `pkg_resources.declare_namespace` so that there are less problems about confusing the `paste` package that is provided by Paste, Paste Script, Paste Deploy, and Paste WebKit. Before you could get one of these at random if you didn't use `pkg_resources.require` first.
* Cleaned up use of `exc_info` argument in `start_response` calls (both accepting and producing), in a variety of places.

CHAPTER 2

The Future Of Paste

2.1 Introduction

Paste has been under development for a while, and has lots of code in it. Too much code! The code is largely decoupled except for some core functions shared by many parts of the code. Those core functions are largely replaced in WebOb, and replaced with better implementations.

The future of these pieces is to split them into independent packages, and refactor the internal Paste dependencies to rely instead on WebOb.

2.2 Already Extracted

paste.fixture: WebTest ScriptTest

paste.lint: wsgiref.validate

paste.exceptions and paste.evalexception: WebError

paste.util.template: Tempita

2.3 To Be Separated

paste.httpserver and paste.debug.watchthreads: Not sure what to call this.

paste.cascade and paste.errordocuments: Not sure; Ben has an implementation of errordocuments in `pylons.middleware.StatusCodeRedirect`

paste.urlmap, paste.deploy.config.PrefixMiddleware: In... some routing thing? Together with the previous package?

paste.proxy: WSGIProxy (needs lots of cleanup though)

paste.fileapp, paste.urlparser.StaticURLParser, paste.urlparser.PkgResourcesParser: In some new file-serving package.

paste.cgiapp, wphp.fcgi_app: Some proxyish app... maybe WSGIProxy?

paste.translogger, paste.debug.prints, paste.util.threadedprint, wsgifilter.proxyapp.DebugHeaders: Some... other place. Something loggy.

paste.registry, paste.config: Not sure. Alberto Valverde expressed interest in splitting out paste.registry.

paste.cgitb_catcher: Move to WebError? Not sure if it matters. For some reason people use this, though.

2.4 To Deprecate

(In that, I won't extract these anywhere; I'm not going to do any big deletes anytime soon, though)

paste.recursive Better to do it manually (with webob.Request.get_response)

paste.wsgiwrappers, paste.request, paste.response, paste.wsgilib, paste.httpheaders, paste.httpexceptions: All the functionality is already in WebOb.

paste.urlparser.URLParser: Really this is tied to paste.webkit more than anything.

paste.auth.*: Well, these all need to be refactored, and replacements exist in AuthKit and repoze.who. Some pieces might still have utility.

paste.debug.profile: I think repoze.profile supersedes this.

paste.debug.wdg_validator: It could get reimplemented with more options for validators, but I'm not really that interested at the moment. The code is nothing fancy.

paste.transaction: More general in repoze.tm

paste.url: No one uses this

2.5 Undecided

paste.debug.fsdiff: Maybe ScriptTest?

paste.session: It's an okay naive session system. But maybe Beaker makes it irrelevant (Beaker does seem slightly more complex to setup). But then, this can just live here indefinitely.

paste.gzipper: I'm a little uncomfortable with this in concept. It's largely in WebOb right now, but not as middleware.

paste.reloader: Maybe this should be moved to paste.script (i.e., paster serve)

paste.debug.debugapp, paste.script.testapp: Alongside other debugging tools, I guess

paste.progress: Not sure this works.

CHAPTER 3

Testing Applications with Paste

author Ian Bicking <ianb@colorstudy.com>

revision $Rev$

date $LastChangedDate$

Contents

- *Testing Applications with Paste*
  - *Introduction*
  - *The Tests Themselves*
  - *Form Submissions*
  - *Framework Hooks*

3.1 Introduction

Paste includes functionality for testing your application in a convenient manner. These facilities are quite young, and feedback is invited. Feedback and discussion should take place on the Paste-users list.

These facilities let you test your Paste and WSGI-based applications easily and without a server.

If you have questions about this document, please contact the paste mailing list or try IRC (`#pythonpaste` on freenode.net). If there's something that confused you and you want to give feedback, please submit an issue.

3.2 The Tests Themselves

The `app` object is a wrapper around your application, with many methods to make testing convenient. Here's an example test script:

```
def test_myapp():
    res = app.get('/view', params={'id': 10})
    # We just got /view?id=10
    res.mustcontain('Item 10')
    res = app.post('/view', params={'id': 10, 'name': 'New item
        name'})
    # The app does POST-and-redirect...
    res = res.follow()
    assert res.request.url == '/view?id=10'
    res.mustcontain('New item name')
    res.mustcontain('Item updated')
```

The methods of the `app` object (a `paste.tests.fixture.TestApp` object):

`get(url, params={}, headers={}, status=None):` Gets the URL. URLs are based in the root of your application; no domains are allowed. Parameters can be given as a dictionary, or included directly in the `url`. Headers can also be added.

This tests that the status is a `200 OK` or a redirect header, unless you pass in a `status`. A status of `"*"` will never fail; or you can assert a specific status (like `500`).

Also, if any errors are written to the error stream this will raise an error.

`post(url, params={}, headers={}, status=None, upload_files=()):` POSTS to the URL. Like GET, except also allows for uploading files. The uploaded files are a list of `(field_name, filename, file_content)`.

If you don't want to do a urlencoded post body, you can put a `content-type` header in your header, and pass the body in as a string with `params`.

The response object:

`header(header_name, [default]):` Returns the named header. It's an error if there is more than one matching header. If you don't provide a default, it is an error if there is no matching header.

`all_headers(header_name):` Returns a list of all matching headers.

`follow(**kw):` Follows the redirect, returning the new response. It is an error if this response wasn't a redirect. Any keyword arguments are passed to `app.get` (e.g., `status`).

`x in res:` Returns True if the string is found in the response. Whitespace is normalized for this test.

`mustcontain(*strings):` Raises an error if any of the strings are not found in the response.

`showbrowser():` Opens the HTML response in a browser; useful for debugging.

`str(res):` Gives a slightly-compacted version of the response.

`click(description=None, linkid=None, href=None, anchor=None, index=None, verbose=False):` Clicks the described link (see docstring for more)

`forms:` Return a dictionary of forms; you can use both indexes (refer to the forms in order) or the string ids of forms (if you've given them ids) to identify the form. See *Form Submissions* for more on the form objects.

Request objects:

`url:` The url requested.

`environ:` The environment used for the request.

`full_url:` The url with query string.

3.3 Form Submissions

You can fill out and submit forms from your tests. First you get the form:

```
res = testapp.get('/entry_form')
form = res.forms[0]
```

Then you fill it in fields:

```
# when there's one unambiguous name field:
form['name'] = 'Bob'
# Enter something into the first field named 'age'
form.set('age', '45', index=1)
```

Finally you submit:

```
# Submit with no particular submit button pressed:
form.submit()
# Or submit a button:
form.submit('submit_button_name')
```

3.4 Framework Hooks

Frameworks can detect that they are in a testing environment by the presence (and truth) of the WSGI environmental variable `"paste.testing"`.

More generally, frameworks can detect that something (possibly a test fixture) is ready to catch unexpected errors by the presence and truth of `"paste.throw_errors"` (this is sometimes set outside of testing fixtures too, when an error-handling middleware is in place).

Frameworks that want to expose the inner structure of the request may use `"paste.testing_variables"`. This will be a dictionary – any values put into that dictionary will become attributes of the response object. So if you do `env["paste.testing_variables"]['template'] = template_name` in your framework, then `response.template` will be `template_name`.

CHAPTER 4

URL Parsing With WSGI And Paste

author Ian Bicking <ianb@colorstudy.com>

revision $Rev$

date $LastChangedDate$

Contents

- *URL Parsing With WSGI And Paste*
  - *Introduction and Audience*
  - *URL Parsing*
  - *Dispatching*
  - *Motivations*
  - *Finding Applications*
  - *Modifying The Request*
  - *Object Publishing*

4.1 Introduction and Audience

This document is intended for web framework authors and integrators, and people who want to understand the internal architecture of Paste.

If you have questions about this document, please contact the paste mailing list or try IRC (`#pythonpaste` on freenode.net). If there's something that confused you and you want to give feedback, please submit an issue.

4.2 URL Parsing

Note: Sometimes people use "URL", and sometimes "URI". I think URLs are a subset of URIs. But in practice you'll almost never see URIs that aren't URLs, and certainly not in Paste. URIs that aren't URLs are abstract Identifiers, that cannot necessarily be used to Locate the resource. This document is *all* about locating.

Most generally, URL parsing is about taking a URL and determining what "resource" the URL refers to. "Resource" is a rather vague term, intentionally. It's really just a metaphor – in reality there aren't any "resources" in HTTP; there are only requests and responses.

In Paste, everything is about WSGI. But that can seem too fancy. There are four core things involved: the *request* (personified in the WSGI environment), the *response* (personified inthe `start_response` callback and the return iterator), the WSGI application, and the server that calls that application. The application and request are objects, while the server and response are really more like actions than concrete objects.

In this context, URL parsing is about mapping a URL to an *application* and a *request*. The request actually gets modified as it moves through different parts of the system. Two dictionary keys in particular relate to URLs – `SCRIPT_NAME` and `PATH_INFO` – but any part of the environment can be modified as it is passed through the system.

4.3 Dispatching

Note: WSGI isn't object oriented? Well, if you look at it, you'll notice there's no objects except built-in types, so it shouldn't be a surprise. Additionally, the interface and promises of the objects we do see are very minimal. An application doesn't have any interface except one method – `__call__` – and that method *does* things, it doesn't give any other information.

Because WSGI is action-oriented, rather than object-oriented, it's more important what we *do*. "Finding" an application is probably an intermediate step, but "running" the application is our ultimate goal, and the only real judge of success. An application that isn't run is useless to us, because it doesn't have any other useful methods.

So what we're really doing is *dispatching* – we're handing the request and responsibility for the response off to another object (another actor, really). In the process we can actually retain some control – we can capture and transform the response, and we can modify the request – but that's not what the typical URL resolver will do.

4.4 Motivations

The most obvious kind of URL parsing is finding a WSGI application.

Typically when a framework first supports WSGI or is integrated into Paste, it is "monolithic" with respect to URLs. That is, you define (in Paste, or maybe in Apache) a "root" URL, and everything under that goes into the framework. What the framework does internally, Paste does not know – it probably finds internal objects to dispatch to, but the framework is opaque to Paste. Not just to Paste, but to any code that isn't in that framework.

That means that we can't mix code from multiple frameworks, or as easily share services, or use WSGI middleware that doesn't apply to the entire framework/application.

An example of someplace we might want to use an "application" that isn't part of the framework would be uploading large files. It's possible to keep track of upload progress, and report that back to the user, but no framework typically is capable of this. This is usually because the POST request is completely read and parsed before it invokes any application code.

This is resolvable in WSGI – a WSGI application can provide its own code to read and parse the POST request, and simultaneously report progress (usually in a way that *another* WSGI application/request can read and report to the user on that progress). This is an example where you want to allow "foreign" applications to be intermingled with framework application code.

4.5 Finding Applications

OK, enough theory. How does a URL parser work? Well, it is a WSGI application, and a WSGI server, in the typical "WSGI middleware" style. Except that it determines which application it will serve for each request.

Let's consider Paste's `URLParser` (in `paste.urlparser`). This class takes a directory name as its only required argument, and instances are WSGI applications.

When a request comes in, the parser looks at `PATH_INFO` to see what's left to parse. `SCRIPT_NAME` represents where we are *now*; it's the part of the URL that has been parsed.

There's a couple special cases:

The empty string:

> URLParser serves directories. When `PATH_INFO` is empty, that means we got a request with no trailing `/`, like say `/blog` If URLParser serves the `blog` directory, then this won't do – the user is requesting the `blog` *page*. We have to redirect them to `/blog/`.

A single `/`:

> So, we got a trailing `/`. This means we need to serve the "index" page. In URLParser, this is some file named `index`, though that's really an implementation detail. You could create an index dynamically (like Apache's file listings), or whatever.

Otherwise we get a string like `/path...`. Note that `PATH_INFO` *must* start with a `/`, or it must be empty.

URLParser pulls off the first part of the path. E.g., if `PATH_INFO` is `/blog/edit/285`, then the first part is `blog`. It appends this to `SCRIPT_NAME`, and strips it off `PATH_INFO` (which becomes `/edit/285`).

It then searches for a file that matches "blog". In URLParser, this means it looks for a filename which matches that name (ignoring the extension). It then uses the type of that file (determined by extension) to create a WSGI application.

One case is that the file is a directory. In that case, the application is *another* URLParser instance, this time with the new directory.

URLParser actually allows per-extension "plugins" – these are just functions that get a filename, and produce a WSGI application. One of these is `make_py` – this function imports the module, and looks for special symbols; if it finds a symbol `application`, it assumes this is a WSGI application that is ready to accept the request. If it finds a symbol that matches the name of the module (e.g., `edit`), then it assumes that is an application *factory*, meaning that when you call it with no arguments you get a WSGI application.

Another function takes "unknown" files (files for which no better constructor exists) and creates an application that simply responds with the contents of that file (and the appropriate `Content-Type`).

In any case, `URLParser` delegates as soon as it can. It doesn't parse the entire path – it just finds the *next* application, which in turn may delegate to yet another application.

Here's a very simple implementation of URLParser:

```
class URLParser(object):
    def __init__(self, dir):
        self.dir = dir
    def __call__(self, environ, start_response):
        segment = wsgilib.path_info_pop(environ)
        if segment is None: # No trailing /
            # do a redirect...
        for filename in os.listdir(self.dir):
            if os.path.splitext(filename)[0] == segment:
                return self.serve_application(
                    environ, start_response, filename)
        # do a 404 Not Found
    def serve_application(self, environ, start_response, filename):
        basename, ext = os.path.splitext(filename)
        filename = os.path.join(self.dir, filename)
        if os.path.isdir(filename):
            return URLParser(filename)(environ, start_response)
        elif ext == '.py':
            module = import_module(filename)
            if hasattr(module, 'application'):
                return module.application(environ, start_response)
            elif hasattr(module, basename):
                return getattr(module, basename)(
                    environ, start_response)
        else:
            return wsgilib.send_file(filename)
```

4.6 Modifying The Request

Well, URLParser is one kind of parser. But others are possible, and aren't too hard to write.

Lets imagine a URL like `/2004/05/01/edit`. It's likely that `/2004/05/01` doesn't point to anything on file, but is really more of a "variable" that gets passed to `edit`. So we can pull them off and put them somewhere. This is a good place for a WSGI extension. Lets put them in `environ["app.url_date"]`.

We'll pass one other applications in – once we get the date (if any) we need to pass the request onto an application that can actually handle it. This "application" might be a URLParser or similar system (that figures out what `/edit` means).

```
class GrabDate(object):
    def __init__(self, subapp):
        self.subapp = subapp
    def __call__(self, environ, start_response):
        date_parts = []
        while len(date_parts) < 3:
           first, rest = wsgilib.path_info_split(environ['PATH_INFO'])
           try:
               date_parts.append(int(first))
               wsgilib.path_info_pop(environ)
           except (ValueError, TypeError):
               break
        environ['app.date_parts'] = date_parts
        return self.subapp(environ, start_response)
```

This is really like traditional "middleware", in that it sits between the server and just one application.

Assuming you put this class in the `myapp.grabdate` module, you could install it by adding this to your configuration:

```
middleware.append('myapp.grabdate.GrabDate')
```

4.7 Object Publishing

Besides looking in the filesystem, "object publishing" is another popular way to do URL parsing. This is pretty easy to implement as well – it usually just means use `getattr` with the popped segments. But we'll implement a rough approximation of Quixote's URL parsing:

```
class ObjectApp(object):
    def __init__(self, obj):
        self.obj = obj
    def __call__(self, environ, start_response):
        next = wsgilib.path_info_pop(environ)
        if next is None:
            # This is the object, lets serve it...
            return self.publish(obj, environ, start_response)
        next = next or '_q_index' # the default index method
        if next in obj._q_export and getattr(obj, next, None):
            return ObjectApp(getattr(obj, next))(
                environ, start_reponse)
        next_obj = obj._q_traverse(next)
        if not next_obj:
            # Do a 404
        return ObjectApp(next_obj)(environ, start_response)

    def publish(self, obj, environ, start_response):
        if callable(obj):
            output = str(obj())
        else:
            output = str(obj)
        start_response('200 OK', [('Content-type', 'text/html')])
        return [output]
```

The `publish` object is a little weak, and functions like `_q_traverse` aren't passed interesting information about the request, but this is only a rough approximation of the framework. Things to note:

- The object has standard attributes and methods – `_q_exports` (attributes that are public to the web) and `_q_traverse` (a way of overriding the traversal without having an attribute for each possible path segment).
- The object isn't rendered until the path is completely consumed (when `next` is `None`). This means `_q_traverse` has to consume extra segments of the path. In this version `_q_traverse` is only given the next piece of the path; Quixote gives it the entire path (as a list of segments).
- `publish` is really a small and lame way to turn a Quixote object into a WSGI application. For any serious framework you'd want to do a better job than what I do here.
- It would be even better if you used something like Adaptation to convert objects into applications. This would include removing the explicit creation of new `ObjectApp` instances, which could also be a kind of fall-back adaptation.

Anyway, this example is less complete, but maybe it will get you thinking.

CHAPTER 5

A Do-It-Yourself Framework

author Ian Bicking <ianb@colorstudy.com>

revision $Rev$

date $LastChangedDate$

This tutorial has been translated into Portuguese.

A newer version of this article is available using WebOb.

Contents

- *A Do-It-Yourself Framework*
  - *Introduction and Audience*
  - *What is WSGI?*
  - *Writing a WSGI Application*
    - *An Interactive App*
  - *Now For a Framework*
    - *Object Publishing*
    - *The Path*
    - *Taking It For a Ride*
    - *Give Me More!*
  - *WSGI Middleware*
    - *Give Me More Middleware!*
  - *Conclusion*

5.1 Introduction and Audience

This short tutorial is meant to teach you a little about WSGI, and as an example a bit about the architecture that Paste has enabled and encourages.

This isn't an introduction to all the parts of Paste – in fact, we'll only use a few, and explain each part. This isn't to encourage everyone to go off and make their own framework (though honestly I wouldn't mind). The goal is that when you have finished reading this you feel more comfortable with some of the frameworks built using this architecture, and a little more secure that you will understand the internals if you look under the hood.

5.2 What is WSGI?

At its simplest WSGI is an interface between web servers and web applications. We'll explain the mechanics of WSGI below, but a higher level view is to say that WSGI lets code pass around web requests in a fairly formal way. But there's more! WSGI is more than just HTTP. It might seem like it is just *barely* more than HTTP, but that little bit is important:

- You pass around a CGI-like environment, which means data like `REMOTE_USER` (the logged-in username) can be securely passed about.
- A CGI-like environment can be passed around with more context – specifically instead of just one path you two: `SCRIPT_NAME` (how we got here) and `PATH_INFO` (what we have left).
- You can – and often should – put your own extensions into the WSGI environment. This allows for callbacks, extra information, arbitrary Python objects, or whatever you want. These are things you can't put in custom HTTP headers.

This means that WSGI can be used not just between a web server an an application, but can be used at all levels for communication. This allows web applications to become more like libraries – well encapsulated and reusable, but still with rich reusable functionality.

5.3 Writing a WSGI Application

The first part is about how to use WSGI at its most basic. You can read the spec, but I'll do a very brief summary:

- You will be writing a *WSGI application*. That's an object that responds to requests. An application is just a callable object (like a function) that takes two arguments: `environ` and `start_response`.
- The environment looks a lot like a CGI environment, with keys like `REQUEST_METHOD`, `HTTP_HOST`, etc.
- The environment also has some special keys like `wsgi.input` (the input stream, like the body of a POST request).
- `start_response` is a function that starts the response – you give the status and headers here.
- Lastly the application returns an iterator with the body response (commonly this is just a list of strings, or just a list containing one string that is the entire body.)

So, here's a simple application:

```
def app(environ, start_response):
    start_response('200 OK', [('content-type', 'text/html')])
    return ['Hello world!']
```

Well... that's unsatisfying. Sure, you can imagine what it does, but you can't exactly point your web browser at it.

There's other cleaner ways to do this, but this tutorial isn't about *clean* it's about *easy-to-understand*. So just add this to the bottom of your file:

```
if __name__ == '__main__':
    from paste import httpserver
    httpserver.serve(app, host='127.0.0.1', port='8080')
```

Now visit http://localhost:8080 and you should see your new app. If you want to understand how a WSGI server works, I'd recommend looking at the CGI WSGI server in the WSGI spec.

5.3.1 An Interactive App

That last app wasn't very interesting. Let's at least make it interactive. To do that we'll give a form, and then parse the form fields:

```
from paste.request import parse_formvars

def app(environ, start_response):
    fields = parse_formvars(environ)
    if environ['REQUEST_METHOD'] == 'POST':
        start_response('200 OK', [('content-type', 'text/html')])
        return ['Hello, ', fields['name'], '!']
    else:
        start_response('200 OK', [('content-type', 'text/html')])
        return ['<form method="POST">Name: <input type="text" '
                '<name="name"><input type="submit"></form>']
```

The `parse_formvars` function just takes the WSGI environment and calls the cgi module (the `FieldStorage` class) and turns that into a MultiDict.

5.4 Now For a Framework

Now, this probably feels a bit crude. After all, we're testing for things like REQUEST_METHOD to handle more than one thing, and it's unclear how you can have more than one page.

We want to build a framework, which is just a kind of generic application. In this tutorial we'll implement an *object publisher*, which is something you may have seen in Zope, Quixote, or CherryPy.

5.4.1 Object Publishing

In a typical Python object publisher you translate `/` to `.`. So `/articles/view?id=5` turns into `root.articles.view(id=5)`. We have to start with some root object, of course, which we'll pass in...

```
class ObjectPublisher(object):

    def __init__(self, root):
        self.root = root

    def __call__(self, environ, start_response):
        ...

app = ObjectPublisher(my_root_object)
```

We override `__call__` to make instances of `ObjectPublisher` callable objects, just like a function, and just like WSGI applications. Now all we have to do is translate that `environ` into the thing we are publishing, then call that thing, then turn the response into what WSGI wants.

5.4.2 The Path

WSGI puts the requested path into two variables: `SCRIPT_NAME` and `PATH_INFO`. `SCRIPT_NAME` is everything that was used up *getting here*. `PATH_INFO` is everything left over – it's the part the framework should be using to find the object. If you put the two back together, you get the full path used to get to where we are right now; this is very useful for generating correct URLs, and we'll make sure we preserve this.

So here's how we might implement `__call__`:

```
def __call__(self, environ, start_response):
    fields = parse_formvars(environ)
    obj = self.find_object(self.root, environ)
    response_body = obj(**fields.mixed())
    start_response('200 OK', [('content-type', 'text/html')])
    return [response_body]

def find_object(self, obj, environ):
    path_info = environ.get('PATH_INFO', '')
    if not path_info or path_info == '/':
        # We've arrived!
        return obj
    # PATH_INFO always starts with a /, so we'll get rid of it:
    path_info = path_info.lstrip('/')
    # Then split the path into the "next" chunk, and everything
    # after it ("rest"):
    parts = path_info.split('/', 1)
    next = parts[0]
    if len(parts) == 1:
        rest = ''
    else:
        rest = '/' + parts[1]
    # Hide private methods/attributes:
    assert not next.startswith('_')
    # Now we get the attribute; getattr(a, 'b') is equivalent
    # to a.b...
    next_obj = getattr(obj, next)
    # Now fix up SCRIPT_NAME and PATH_INFO...
    environ['SCRIPT_NAME'] += '/' + next
    environ['PATH_INFO'] = rest
    # and now parse the remaining part of the URL...
    return self.find_object(next_obj, environ)
```

And that's it, we've got a framework.

5.4.3 Taking It For a Ride

Now, let's write a little application. Put that `ObjectPublisher` class into a module `objectpub`:

```
from objectpub import ObjectPublisher

class Root(object):

    # The "index" method:
    def __call__(self):
        return '''
        <form action="welcome">
        Name: <input type="text" name="name">
        <input type="submit">
        </form>
        '''

    def welcome(self, name):
        return 'Hello %s!' % name

app = ObjectPublisher(Root())
```

```
if __name__ == '__main__':
    from paste import httpserver
    httpserver.serve(app, host='127.0.0.1', port='8080')
```

Alright, done! Oh, wait. There's still some big missing features, like how do you set headers? And instead of giving `404 Not Found` responses in some places, you'll just get an attribute error. We'll fix those up in a later installment...

5.4.4 Give Me More!

You'll notice some things are missing here. Most specifically, there's no way to set the output headers, and the information on the request is a little slim.

```
# This is just a dictionary-like object that has case-
# insensitive keys:
from paste.response import HeaderDict

class Request(object):
    def __init__(self, environ):
        self.environ = environ
        self.fields = parse_formvars(environ)

class Response(object):
    def __init__(self):
        self.headers = HeaderDict(
            {'content-type': 'text/html'})
```

Now I'll teach you a little trick. We don't want to change the signature of the methods. But we can't put the request and response objects in normal global variables, because we want to be thread-friendly, and all threads see the same global variables (even if they are processing different requests).

But Python 2.4 introduced a concept of "thread-local values". That's a value that just this one thread can see. This is in the threading.local object. When you create an instance of `local` any attributes you set on that object can only be seen by the thread you set them in. So we'll attach the request and response objects here.

So, let's remind ourselves of what the `__call__` function looked like:

```
class ObjectPublisher(object):
    ...

    def __call__(self, environ, start_response):
        fields = parse_formvars(environ)
        obj = self.find_object(self.root, environ)
        response_body = obj(**fields.mixed())
        start_response('200 OK', [('content-type', 'text/html')])
        return [response_body]
```

Lets's update that:

```
import threading
webinfo = threading.local()

class ObjectPublisher(object):
    ...

    def __call__(self, environ, start_response):
        webinfo.request = Request(environ)
        webinfo.response = Response()
        obj = self.find_object(self.root, environ)
        response_body = obj(**dict(webinfo.request.fields))
        start_response('200 OK', webinfo.response.headers.items())
        return [response_body]
```

Now in our method we might do:

```
class Root:
    def rss(self):
        webinfo.response.headers['content-type'] = 'text/xml'
        ...
```

If we were being fancier we would do things like handle cookies in these objects. But we aren't going to do that now. You have a framework, be happy!

5.5 WSGI Middleware

Middleware is where people get a little intimidated by WSGI and Paste.

What is middleware? Middleware is software that serves as an intermediary.

So lets write one. We'll write an authentication middleware, so that you can keep your greeting from being seen by just anyone.

Let's use HTTP authentication, which also can mystify people a bit. HTTP authentication is fairly simple:

- When authentication is requires, we give a `401 Authentication Required` status with a `WWW-Authenticate: Basic realm="This Realm"` header
- The client then sends back a header `Authorization: Basic encoded_info`
- The "encoded_info" is a base-64 encoded version of `username:password`

So how does this work? Well, we're writing "middleware", which means we'll typically pass the request on to another application. We could change the request, or change the response, but in this case sometimes we *won't* pass the request on (like, when we need to give that 401 response).

To give an example of a really really simple middleware, here's one that capitalizes the response:

```
class Capitalizer(object):

    # We generally pass in the application to be wrapped to
    # the middleware constructor:
    def __init__(self, wrap_app):
        self.wrap_app = wrap_app

    def __call__(self, environ, start_response):
        # We call the application we are wrapping with the
        # same arguments we get...
        response_iter = self.wrap_app(environ, start_response)
        # then change the response...
        response_string = ''.join(response_iter)
        return [response_string.upper()]
```

Techically this isn't quite right, because there there's two ways to return the response body, but we're skimming bits. paste.wsgilib.intercept_output is a somewhat more thorough implementation of this.

Note: This, like a lot of parts of this (now fairly old) tutorial is better, more thorough, and easier using WebOb. This particular example looks like:

```
from webob import Request

class Capitalizer(object):
    def __init__(self, app):
        self.app = app
    def __call__(self, environ, start_response):
        req = Request(environ)
        resp = req.get_response(self.app)
        resp.body = resp.body.upper()
        return resp(environ, start_response)
```

So here's some code that does something useful, authentication:

```
class AuthMiddleware(object):

    def __init__(self, wrap_app):
        self.wrap_app = wrap_app

    def __call__(self, environ, start_response):
        if not self.authorized(environ.get('HTTP_AUTHORIZATION')):
            # Essentially self.auth_required is a WSGI application
            # that only knows how to respond with 401...
            return self.auth_required(environ, start_response)
        # But if everything is okay, then pass everything through
        # to the application we are wrapping...
        return self.wrap_app(environ, start_response)

    def authorized(self, auth_header):
        if not auth_header:
            # If they didn't give a header, they better login...
            return False
        # .split(None, 1) means split in two parts on whitespace:
        auth_type, encoded_info = auth_header.split(None, 1)
        assert auth_type.lower() == 'basic'
        unencoded_info = encoded_info.decode('base64')
        username, password = unencoded_info.split(':', 1)
        return self.check_password(username, password)

    def check_password(self, username, password):
        # Not very high security authentication...
        return username == password

    def auth_required(self, environ, start_response):
        start_response('401 Authentication Required',
            [('Content-type', 'text/html'),
             ('WWW-Authenticate', 'Basic realm="this realm"')])
        return ["""
        <html>
         <head><title>Authentication Required</title></head>
         <body>
          <h1>Authentication Required</h1>
          If you can't get in, then stay out.
         </body>
        </html>"""]
```

Note: Again, here's the same thing with WebOb:

```
from webob import Request, Response

class AuthMiddleware(object):
    def __init__(self, app):
        self.app = app
    def __call__(self, environ, start_response):
        req = Request(environ)
        if not self.authorized(req.headers['authorization']):
            resp = self.auth_required(req)
        else:
            resp = self.app
        return resp(environ, start_response)
    def authorized(self, header):
        if not header:
            return False
        auth_type, encoded = header.split(None, 1)
        if not auth_type.lower() == 'basic':
            return False
        username, password = encoded.decode('base64').split(':', 1)
        return self.check_password(username, password)
 def check_password(self, username, password):
     return username == password
 def auth_required(self, req):
     return Response(status=401, headers={'WWW-Authenticate': 'Basic realm="this realm"'},
                     body="""\
     <html>
      <head><title>Authentication Required</title></head>
      <body>
       <h1>Authentication Required</h1>
       If you can't get in, then stay out.
      </body>
     </html>""")
```

So, how do we use this?

```
app = ObjectPublisher(Root())
wrapped_app = AuthMiddleware(app)

if __name__ == '__main__':
    from paste import httpserver
    httpserver.serve(wrapped_app, host='127.0.0.1', port='8080')
```

Now you have middleware! Hurrah!

5.5.1 Give Me More Middleware!

It's even easier to use other people's middleware than to make your own, because then you don't have to program. If you've been following along, you've probably encountered a few exceptions, and have to look at the console to see the exception reports. Let's make that a little easier, and show the exceptions in the browser...

```
app = ObjectPublisher(Root())
wrapped_app = AuthMiddleware(app)
from paste.exceptions.errormiddleware import ErrorMiddleware
exc_wrapped_app = ErrorMiddleware(wrapped_app)
```

Easy! But let's make it *more* fancy...

```
app = ObjectPublisher(Root())
wrapped_app = AuthMiddleware(app)
from paste.evalexception import EvalException
exc_wrapped_app = EvalException(wrapped_app)
```

So go make an error now. And hit the little +'s. And type stuff in to the boxes.

5.6 Conclusion

Now that you've created your framework and application (I'm sure it's much nicer than the one I've given so far). You might keep writing it (many people have so far), but even if you don't you should be able to recognize these components in other frameworks now, and you'll have a better understanding how they probably work under the covers.

Also check out the version of this tutorial written using WebOb. That tutorial includes things like **testing** and **pattern-matching dispatch** (instead of object publishing).

CHAPTER 6

The Paste HTTP Server Thread Pool

This document describes how the thread pool in `paste.httpserver` works, and how it can adapt to problems.

Note all of the configuration parameters listed here are prefixed with `threadpool_` when running through a Paste Deploy configuration.

6.1 Error Cases

When a WSGI application is called, it's possible that it will block indefinitely. There's two basic ways you can manage threads:

- Start a thread on every request, close it down when the thread stops
- Start a pool of threads, and reuse those threads for subsequent requests

In both cases things go wrong – if you start a thread every request you will have an explosion of threads, and with it memory and a loss of performance. This can culminate in really high loads, swapping, and the whole site grinds to a halt.

If you are using a pool of threads, all the threads can simply be used up. New requests go into a queue to be processed, but since that queue never moves forward everyone will just block. The site basically freezes, though memory usage doesn't generally get worse.

6.2 Paste Thread Pool

The thread pool in Paste has some options to walk the razor's edge between the two techniques, and to try to respond usefully in most cases.

The pool tracks all workers threads. Threads can be in a few states:

- Idle, waiting for a request ("idle")
- Working on a request
  - For a reasonable amount of time ("busy")
  - For an unreasonably long amount of time ("hung")
- Thread that should die
  - An exception has been injected that should kill the thread, but it hasn't happened yet ("dying")
  - An exception has been injected, but the thread has persisted for an unreasonable amount of time ("zombie")

When a request comes in, if there are no idle worker threads waiting then the server looks at the workers; all workers are busy or hung. If too many are hung, another thread is opened up. The limit is if there are less than `spawn_if_under` busy threads. So if you have 10 workers, `spawn_if_under` is 5, and there are 6 hung threads and 4 busy threads, another thread will be opened (bringing the number of busy threads back to 5). Later those threads may be collected again if some of the threads become un-hung. A thread is hung if it has been working for longer than `hung_thread_limit` (default 30 seconds).

Every so often, the server will check all the threads for error conditions. This happens every `hung_check_period` requests (default 100). At this time if there are more than enough threads (because of `spawn_if_under`) some threads may be collected. If any threads have been working for longer than `kill_thread_limit` (default 1800 seconds, i.e., 30 minutes) then the thread will be killed.

To kill a thread the `ctypes` module must be installed. This will raise an exception (`SystemExit`) in the thread, which should cause the thread to stop. It can take quite a while for this to actually take effect, sometimes on the order of several minutes. This uses a non-public API (hence the `ctypes` requirement), and so it might not work in all cases. I've tried it in pure Python code and with a hung socket, and in both cases it worked. As soon as the thread is killed (before it is actually dead) another worker is added to the pool.

If the killed thread lives longer than `dying_thread_limit` (default 300 seconds, 5 minutes) then it is considered a zombie.

Zombie threads are not handled specially unless you set `max_zombies_before_die`. If you set this and there are more than this many zombie threads, then the entire process will be killed. This is useful if you are running the server under some process monitor, such as `start-stop-daemon`, `daemontools`, `runit`, or with `paster serve --monitor`. To make the process die, it may run `os._exit`, which is considered an impolite way to exit a process (akin to `kill -9`). It *will* try to run the functions registered with `atexit` (except for the thread cleanup functions, which are the ones which will block so long as there are living threads).

6.3 Notification

If you set `error_email` (including setting it globally in a Paste Deploy `[DEFAULT]` section) then you will be notified of two error conditions: when hung threads are killed, and when the process is killed due to too many zombie threads.

6.4 Missed Cases

If you have a worker pool size of 10, and 11 slow or hung requests come in, the first 10 will get handed off but the server won't know yet that they will hang. The last request will stay stuck in a queue until another request comes in. When a later request comes later (after `hung_thread_limit` seconds) the server will notice the problem and add more threads, and the 11th request will come through.

If a trickle of bad requests keeps coming in, the number of hung threads will keep increasing. At 100 the `hung_check_period` may not clean them up fast enough.

Killing threads is not something Python really supports. Corruption of the process, memory leaks, or who knows what might occur. For the most part the threads seem to be killed in a fairly simple manner – an exception is raised, and `finally` blocks do get executed. But this hasn't been tried much in production, so there's not much experience with it.

6.5 watch_threads

If you want to see what's going on in your process, you can install the application `egg:Paste#watch_threads` (in the `paste.debug.watchthreads` module). This lets you see requests and how long they have been running. In Python 2.5 you can see tracebacks of the running requests; before that you can only see request data (URLs, User-Agent, etc). If you set `allow_kill = true` then you can also kill threads from the application. The thread pool is intended to run reliably without intervention, but this can help debug problems or give you some feeling of what causes problems in the site.

This does open up privacy problems, as it gives you access to all the request data in the site, including cookies, IP addresses, etc. It shouldn't be left on in a public setting.

6.6 socket_timeout

The HTTP server (not the thread pool) also accepts an argument `socket_timeout`. It is turned off by default. You might find it helpful to turn it on.

CHAPTER 7

Features

7.1 Testing

- A fixture for testing WSGI applications conveniently and in-process, in `paste.fixture.TestApp`
- A fixture for testing command-line applications, also in `paste.fixture.TestFileEnvironment`
- Check components for WSGI-compliance in `paste.lint`
- Check filesystem changes, with `paste.debug.fsdiff`

7.2 Server

- A threaded HTTP server in `paste.httpserver`
- A tool for seeing and killing errant threads in the HTTP server, in `paste.debug.watchthreads`

7.3 Dispatching

- Chain and cascade WSGI applications (returning the first non-error response) in `paste.cascade`
- Dispatch to several WSGI applications based on URL prefixes, in `paste.urlmap`
- Allow applications to make subrequests and forward requests internally, in `paste.recursive`
- Redirect error pages (e.g., 404 Not Found) to custom error pages, in `paste.errordocument`.

7.4 Web Application

- Easily deal with incoming requests and sending a response in `paste.wsgiwrappers`
- Work directly with the WSGI environment in `paste.request`
- Run CGI programs as WSGI applications in `paste.cgiapp`
- Traverse files and load WSGI applications from `.py` files (or static files), in `paste.urlparser`
- Serve static directories of files, also in `paste.urlparser`; also serve using the Setuptools `pkg_resources` resource API.

- Proxy to other servers, treating external HTTP servers as WSGI applications, in `paste.proxy`.
- Serve files (with support for `If-Modified-Since`, etc) in `paste.fileapp`

7.5 Tools

- Catch HTTP-related exceptions (e.g., `HTTPNotFound`) and turn them into proper responses in `paste.httpexceptions`
- Manage HTTP header fields with `paste.httpheaders`
- Handle authentication/identification of requests in `paste.auth`
- Create sessions in `paste.session` and `paste.flup_session`
- Gzip responses in `paste.gzipper`
- A wide variety of routines for manipulating WSGI requests and producing responses, in `paste.request`, `paste.response` and `paste.wsgilib`.
- Create Apache-style logs in `paste.translogger`
- Handy request and response wrappers in `paste.wsgiwrappers`
- Handling of request-local module globals sanely in `paste.registry`

7.6 Authentication

- Authentication using cookies in `paste.auth.cookie` and `paste.auth.auth_tkt`; login form in `paste.auth.form`
- Authentication using OpenID in `paste.auth.open_id`, using CAS in `paste.auth.cas`
- HTTP authentication in `paste.auth.basic` and `paste.auth.digest`
- Dispatch to different authentication methods based on User-Agent, in `paste.auth.multi`
- Grant roles based on IP addresses, in `paste.auth.grantip`

7.7 Debugging Filters

- Catch (optionally email) errors with extended tracebacks (using Zope/ZPT conventions) in `paste.exceptions`
- During debugging, show tracebacks with information about each stack frame, including an interactive prompt that runs in the individual stack frames, in `paste.evalexception`.
- Catch errors presenting a cgitb-based output, in `paste.cgitb_catcher`.
- Profile each request and append profiling information to the HTML, in `paste.debug.profile`
- Capture `print` output and present it in the browser for debugging, in `paste.debug.prints`
- Validate all HTML output from applications using the WDG Validator, appending any errors or warnings to the page, in `paste.debug.wdg_validator`

7.8 Other Tools

- A file monitor to allow restarting the server when files have been updated (for automatic restarting when editing code) in `paste.reloader`
- A class for generating and traversing URLs, and creating associated HTML code, in `paste.url`
- A small templating language (for internal use) in `paste.util.template`
- A class to help with loops in templates, in `paste.util.looper`
- Import modules and objects given a string, in `paste.util.import_string`
- Ordered dictionary that can have multiple values with the same key, in `paste.util.multidict`

CHAPTER 8

Python Paste Developer Guide

Hi. Welcome to Paste. I hope you enjoy your stay here.

I hope to bring together multiple efforts here, for Paste to support multiple frameworks and directions, while presenting a fairly integrated frontend to users. How to do that? That's an open question, and this code is in some ways an exploration.

There's some basic principles:

- Keep stuff decoupled.
- Must be testable. Of course tested is even better than testable.
- Use WSGI standards for communication between decoupled libraries.
- When possible, use HTTP semantics for communicating between libraries (e.g., indicate cachability using the appropriate HTTP headers).
- When possible, use WSGI as a wrapper around web-neutral libraries. For instance, the configuration is a simple library, but the WSGI middleware that puts the configuration in place is really really simple. If it could be used outside of a web context, then having both a library and middleware form is good.
- Entry into frameworks should be easy, but exit should also be easy. Heterogeneous frameworks and applications are the ambition. But we have to get some messiness into Paste before we can try to resolve that messiness.
- When all is said and done, users should be able to ignore much of what we've done and focus on writing their applications, and Stuff Just Works. Documentation is good; stuff that works without user intervention is better.

8.1 Developer Info

Mostly, if there's a problem we can discuss it and work it out, no one is going to bite your head off for committing something.

- Framework-like things should go in subpackages, or perhaps in separate distributions entirely (Paste WebKit and Wareweb were extracted for this reason).
- Integrating external servers and frameworks is also interesting, but it's best to introduce that as a requirement instead of including the work here. Paste Script contains several wrappers for external projects (servers in particular).
- Tests are good. We use py.test, because it is simple. I want to use doctests too, but the infrastructure isn't really there now – but please feel free to use those too. `unittest` is kind of annoying, and py.test is both more powerful and easier to write for. Tests should go in the `tests/` directory. `paste.fixture` contains some convenience functions for testing WSGI applications and middleware. Pay particular attention to `TestApp`.

- If you move something around that someone may be using, keep their imports working and introduce a warning, like:

```
def backward_compat_function(*args, **kw):
    import warnings
    # Deprecated on 2005 Mar 5
    warnings.warn('Moved to foo.function', DeprecationWarning, 2)
    return foo.function(*args, **kw)
```

- If something is really experimental, put it in your home directory, or make a branch in your home directory. You can make a home directory for yourself, in `http://svn.w4py.org/home/username`.
- Not everything in the repository or even in the trunk will necessarily go into the release. The release should contain stuff that is tested, documented, and useful. Each module or feature also needs a champion – someone who will stand by the code, answer questions, etc. It doesn't have to be the original developer, but there has to be *someone*. So when a release is cut, if some modules don't fulfill that they may be left out.
- Try to keep to the Style Guidelines. But if you are bringing in outside work, don't stress out too much about it. Still, if you have a choice, follow that. Those guidelines are meant to represent conventional Python style guides, there's nothing out of the normal there.
- Write your docstrings in restructured text. As time goes on, I want to rely on docstrings more for documentation, with shorter narrative documentation pointing into the documentation generated from docstrings.

  The generation is done with Pudge. To try generating the documentation, this should work:

```
$ easy_install svn://lesscode.org/buildutils/trunk \
               svn://lesscode.org/pudge/trunk
$ cd Paste
$ python setup.py pudge
```

  This will install Pudge and buildutils, and then generate the documentation into `Paste/docs/html/`.

CHAPTER 9

Paste Style Guide

Generally you should follow the recommendations in PEP 8, the Python Style Guide. Some things to take particular note of:

- **No tabs**. Not anywhere. Always indent with 4 spaces.
- I don't stress too much on line length. But try to break lines up by grouping with parenthesis instead of with backslashes (if you can). Do asserts like:

```
assert some_condition(a, b), (
    "Some condition failed, %r isn't right!" % a)
```

- But if you are having problems with line length, maybe you should just break the expression up into multiple statements.
- Blank lines between methods, unless they are very small and closely bound to each other.
- Don't use the form `condition and trueValue or falseValue`. Break it out and use a variable.
- I (Ian Bicking) am very picky about whitespace. There's one and only one right way to do it. Good examples:

```
short = 3
longerVar = 4

if x == 4:
    do stuff

func(arg1='a', arg2='b')
func((a + b)*10)
```

  Bad examples:

```
short    =3
longerVar=4

if x==4: do stuff

func(arg1 = 'a', arg2 = 'b')
func(a,b)
func( a, b )
[ 1, 2, 3 ]
```

  If the whitespace isn't right, it'll annoy me and I'll feel a need to fix it. Really, this whitespace stuff shouldn't be that controversial should it? Some particular points that I feel strongly about:

  – No space after a function name (bad: `func (arg)`).

  - No space after or before a parenthesis (bad: `func( arg )`).
  - Always one space after a comma (bad: `func(a,b)`).
- Use `@@` to mark something that is suboptimal, or where you have a concern that it's not right. Try to also date it and put your username there.
- Docstrings are good. They should look like:

```
class AClass(object):
    """
    doc string...
    """
```

  Don't use single quotes ('‘‘) – they tend to cause problems in Emacs. Don't bother trying make the string less vertically compact.

- Comments go right before the thing they are commenting on.
- Methods never, ever, ever start with capital letters. Generally only classes are capitalized. But definitely never methods.
- Use `cls` to refer to a class. Use `meta` to refer to a metaclass (which also happens to be a class, but calling a metaclass `cls` will be confusing).
- Use `isinstance` instead of comparing types. E.g.:

```
if isinstance(var, str): ...
# Bad:
if type(var) is StringType: ...
```

- Never, ever use two leading underscores. This is annoyingly private. If name clashes are a concern, use explicit name mangling instead (e.g., `_MyThing_blahblah`). This is essentially the same thing as double-underscore, only it's transparent where double underscore obscures.
- Module names should be unique in the package. Subpackages shouldn't share module names with sibling or parent packages. Sadly this isn't possible for `__init__.py`, but it's otherwise easy enough.
- Module names should be all lower case, and probably have no underscores (smushedwords).

CHAPTER 10

Community

Much of the communication goes on in the mailing lists; see that page for information on the lists.

For live IRC discussion, try the `#pythonpaste` channel on Freenode.

If you find bugs in the code or documentation, please submit a ticket. You can also view tickets.

CHAPTER 11

Mailing Lists

General discussion and questions should go to:

paste-users@googlegroups.org: New posts are on Google Groups old posts are in their own archive

More abstract discussion of Python web programming should go to:

web-sig@python.org: Subscribe, Archives

CHAPTER 12

Repository

Paste is kept in a Mercurial (hg) repository at http://bitbucket.org/ianb/paste

If you are using a command-line Mercurial client, you can check it out like:

```
hg clone http://bitbucket.org/ianb/paste
```

CHAPTER 13

Downloads

Each of these packages is available in several formats. The source distribution is a complete set of documentation, tests, and the source files themselves. There are also two "Egg" files: these are files easy_install can install directly into your `site-packages/` directory, and are Python-version specific. The download files for the latest version are always located on the Cheese Shop pages (listed below).

- Paste
- Paste Script
- Paste Deploy
- Paste WebKit
- Wareweb (deprecated, use Pylons instead)

All the packages are available in the Mercurial repositories rooted in http://bitbucket.org/ianb/

- http://bitbucket.org/ianb/paste
- http://bitbucket.org/ianb/pastescript
- http://bitbucket.org/ianb/pastedeploy
- https://github.com/Pylons/webob
- ... and others

Use:

```
hg clone http://bitbucket.org/ianb/paste
```

to check out a working copy of Paste.

CHAPTER 14

Indices and tables

- genindex
- modindex
- search

CHAPTER 15

Other Components

- Paste Deploy
- Paste Script
- WebOb
- WSGI specification (PEP 333)

CHAPTER 16

License

Paste is distributed under the MIT license.