wheezy.http documentation

Release latest

Andriy Kornatskyy

Apr 17, 2021

Contents

CHAPTER 1

Introduction

wheezy.http is a python package written in pure Python code. It is a lightweight http library for things like request, response, headers, cookies and many others. It a wrapper around the WSGI request environment.

It is optimized for performance, well tested and documented.

Resources:

- source code, examples and issues tracker are available on github
- documentation

CHAPTER 2

Contents

2.1 Getting Started

2.1.1 Install

wheezy.http requires python version 3.6+. It is operating system independent. You can install it from pypi:

```
$ easy_install wheezy.http
```

2.2 Examples

Before we proceed let's setup a virtualenv environment, activate it and install:

```
$ pip install wheezy.http
```

2.2.1 Hello World

helloworld.py shows you how to use *wheezy.http* in a pretty simple WSGI application:

```
    HTTPResponse,
    WSGIApplication,
    bootstrap_http_defaults,
    not_found,
)


def welcome(request):
    response = HTTPResponse()
    response.write("Hello World!!!")
```

(continues on next page)

(continued from previous page)

```
    return response


def router_middleware(request, following):
    path = request.path
    if path == "/":
        response = welcome(request)
    else:
        response = not_found()
    return response


options = {}
main = WSGIApplication(
    [bootstrap_http_defaults, lambda ignore: router_middleware], options
)


if __name__ == "__main__":
    from wsgiref.simple_server import make_server

    try:
        print("Visit http://localhost:8080/")
        make_server("", 8080, main).serve_forever()
    except KeyboardInterrupt:
        pass
    print("\nThanks!")
```

Let have a look through each line in this application.

Request Handler

First of all let's take a look what is request handler:

```
def welcome(request):
    response = HTTPResponse()
    response.write("Hello World!!!")
```

It is a simple callable of form:

```
def handler(request):
    return response
```

In *wheezy.http* there are no dependencies between `HTTPRequest` and `HTTPResponse`.

While *wheezy.http* doesn't prescribe what is a router, we add here a simple router middleware. This way you can use one of available alternatives to provide route matching for your application.

```
def router_middleware(request, following):
    path = request.path
    if path == "/":
        response = welcome(request)
    else:
        response = not_found()
```

There is a separate python package wheezy.routing that is recommended way to add routing facility to your application.

Finally we create the entry point that is an instance of `WSGIApplication`.

```
options = {}
main = WSGIApplication(
    [bootstrap_http_defaults, lambda ignore: router_middleware], options
)
```

The rest in the `helloworld` application launches a simple wsgi server. Try it by running:

```
$ python helloworld.py
```

Visit http://localhost:8080/.

2.2.2 Guest Book

TODO

2.3 User Guide

wheezy.http is a lightweight WSGI library that aims to take most benefits out of standard python library. It can be run from python 2.4 up to most cutting age python 3.

2.3.1 Configuration Options

Configuration options is a python dictionary passed to `WSGIApplication` during initialization. These options are shared across various parts of application, including: middleware factory, http request, cookies, etc.

```
options = {}
main = WSGIApplication(
    [bootstrap_http_defaults, lambda ignore: router_middleware], options
)
```

There are no required options necessarily setup before use, since they all fallback to some defaults defined in the `config` module. Actually `options` are checked by the `bootstrap_http_defaults()` middleware factory for missing values (the middleware factory is executed only once at application start up).

See full list of available options in `config` module.

2.3.2 WSGI Application

WSGI is the Web Server Gateway Interface. It is a specification for web/application servers to communicate with web applications. It is a Python standard, described in detail in PEP 3333.

An instance of `WSGIApplication` is an entry point of your WSGI application. You instantiate it by supplying a list of desired `middleware factories` and global configuration `options`. Here is a snippet from *Hello World* example:

```
options = {}
main = WSGIApplication(
    [bootstrap_http_defaults, lambda ignore: router_middleware], options
)
```

An instance of `WSGIApplication` is a callable that responds to the standard WSGI call. This callable is passed to application/web server. Here is an integration example with the web server from python standard `wsgiref` package:

```
    from wsgiref.simple_server import make_server

    try:
        print("Visit http://localhost:8080/")
        make_server("", 8080, main).serve_forever()
    except KeyboardInterrupt:
        pass
```

The integration with other WSGI application servers varies. However the principal of WSGI entry point is the same across those implementations.

2.3.3 Middleware

The presence of middleware, in general, is transparent to the application and requires no special support. Middleware is usually characterized by playing the following roles within an application:

- It is singleton, there is only one instance per application.
- It is sort of interceptor of incoming request to handler.
- They can be chained so one pass request to following as well as capable to inspect response, override it, extend or modify as necessary.
- It capable to supply additional information in request context.

Middleware can be any callable of the following form:

```
def middleware(request, following):
    if following is not None:
        response = following(request)
    else:
        response = ...
    return response
```

A middleware callable accepts as a first argument an instance of `HTTPRequest` and as second argument (`following`) the next middleware in the chain. It is up to middleware to decide whether to call the next middleware callable in the chain. It is expected that middleware returns an instance of `HTTPResponse` class or `None`.

Middleware Factory

Usually middleware requires some sort of initialization before being used. This can be some configuration variables or sort of preparation, verification, etc. `Middleware Factory` serves this purpose.

Middleware factory can be any callable of the following form:

```
def middleware_factory(options):
    return middleware
```

Middleware factory is initialized with configuration `options`, it is the same dictionary used during `WSGIApplication` initialization. Middleware factory returns particular middleware implementation or `None` (this can be useful for some sort of initialization that needs to be run during application bootstrap, e.g. some defaults, see `bootstrap_http_defaults()`).

In case the last middleware in the chain returns `None` it is equivalent to returning HTTP response not found (HTTP status code 404).

Execution Order

Middleware is initialized and executed in certain order. Let's setup a simple application with the following middleware chain:

```
app = WSGIApplication(middleware=[
    a_factory,
    b_factory,
    c_factory
])
```

Initialization and execution order is the same - from first element in the list to the last:

```
a_factory => b_factory => c_factory
```

In case a factory returns `None` it is being skipped from middleware list. Let assume `b_factory` returns `None`, so the middleware chain become:

```
a => c
```

It is up to middleware `a` to call `c` before or after its own processing. `WSGIApplication` in no way prescribes it, instead it just chains them. This gives great power to the middleware developer to take control over certain implementation use case.

2.3.4 HTTP Handler

Handler is any callable that accepts an instance of `HTTPRequest` and returns `HTTPResponse`:

```
def handler(request):
    return response
```

Here is an example:

```
def welcome(request):
    response = HTTPResponse()
    response.write("Hello World!!!")
```

wheezy.http does not provide HTTP handler implementations (see wheezy.web for this purpose).

@accept_method

Decorator `accept_method` accepts only particular HTTP request method if its argument (`constraint`) is a string:

```
@accept_method('GET')
def my_view(request):
    ...
```

or one of multiple HTTP request methods if the argument (`constraint`) is a list or tuple:

```
@accept_method(('GET', 'POST'))
def my_view(request):
    ...
```

Method argument constraint must be in uppercase.

Respond with an HTTP status code 405 (Method Not Allowed) in case incoming HTTP request method does not match decorator constraint.

@secure

Decorator `secure` accepts only secure requests (those that are communication via SSL):

```
@secure
def my_view(request):
    ...
```

Its behavior can be controlled via `enabled` (in case it is `False` no checks are performed, defaults to `True`).

2.3.5 HTTP Request

`HTTPRequest` is a wrapper around WSGI environ dictionary. It provides access to all variables stored within the environ as well as provide several handy methods for daily use.

`HTTPRequest` includes the following useful attributes (they are evaluated only once during processing):

- `method` - request method (GET, POST, HEAD, etc)
- `host` - request host; depends on WSGI variable `HTTP_HOST`.
- `remote_addr` - remote address; depends on WSGI variable `REMOTE_ADDR`.
- `root_path` - application virtual path; environ `SCRIPT_NAME` plus `/`.
- `path` - request url path; environ `SCRIPT_NAME` plus `PATH_INFO`.
- `query` - request url query; data are returned as a dictionary. The dictionary keys are the unique query variable names and the values are lists of values for each name. Supports a compact form in which the param may be given once but set to a list of comma-separated values (e.g., 'id=1,2,3').
- `form` - request form; data are returned as a dictionary. The dictionary keys are the unique form variable names and the values are lists of values for each name. Supports the following mime types: `application/x-www-form-urlencoded`, `application/json` and `multipart/form-data`.
- `files` - request form files; data are returned as a dictionary. The dictionary keys are the unique file variable names and the values are lists of files (`cgi.FieldStorage`) for each name.
- `cookies` - cookies passed by browser; an instance of `dict`.
- `ajax` - returns `True` if current request is AJAX request.
- `secure` - determines whether the current request was made via SSL connection; depends on WSGI variable `wsgi.url_scheme`.

- `scheme` - request url scheme (`http` or `https`); depends on WSGI variable `wsgi.url_scheme`.
- `urlparts` - returns a tuple of 5, corresponding to request url: scheme, host, path, query and fragment (always `None`).
- `content_type` - returns the MIME content type of the incoming request.
- `content_length` - returns the length, in bytes, of content sent by the client.
- `stream` - returns the contents of the incoming HTTP entity body.

Form and Query

While working with request form/query you get a dictionary. The dictionary keys are the unique form variable names and the values are lists of values for each name. There usually exists just one value, so working with list is not that convenient. You can use `get_param` or `first_item_adapter` or `last_item_adapter` (see wheezy.core):

```
>>> from wheezy.core.collections import last_item_adapter
...
>>> request.query['a']
['1', '2']
>>> query = last_item_adapter(request.query)
>>> query['a']
'2'
>>> request.get_param('a')
'2'
```

While you are able initialize your application models by requesting certain values from `form` or `query`, there is a separate python package wheezy.validation that is recommended way to add forms facility to your application. It includes both model binding as well as a number of validation rules.

Supported content types: *application/x-www-form-urlencoded*, *application/json* and *multipart/form-data*.

2.3.6 HTTP Response

`HTTPResponse` correctly maps the following HTTP response status codes (according to rfc2616):

```
HTTP_STATUS = {
    # Informational
    100: "100 Continue",
    101: "101 Switching Protocols",
    # Successful
    200: "200 OK",
    201: "201 Created",
    202: "202 Accepted",
    203: "203 Non-Authoritative Information",
    204: "204 No Content",
    205: "205 Reset Content",
    206: "206 Partial Content",
    207: "207 Multi-Status",
    # Redirection
    300: "300 Multiple Choices",
    301: "301 Moved Permanently",
    302: "302 Found",
    303: "303 See Other",
    304: "304 Not Modified",
    305: "305 Use Proxy",
```

(continues on next page)

(continued from previous page)

```
    307: "307 Temporary Redirect",
    # Client Error
    400: "400 Bad Request",
    401: "401 Unauthorized",
    402: "402 Payment Required",
    403: "403 Forbidden",
    404: "404 Not Found",
    405: "405 Method Not Allowed",
    406: "406 Not Acceptable",
    407: "407 Proxy Authentication Required",
    408: "408 Request Timeout",
    409: "409 Conflict",
    410: "410 Gone",
    411: "411 Length Required",
    412: "412 Precondition Failed",
    413: "413 Request Entity Too Large",
    414: "414 Request-Uri Too Long",
    415: "415 Unsupported Media Type",
    416: "416 Requested Range Not Satisfiable",
    417: "417 Expectation Failed",
    # Server Error
    500: "500 Internal Server Error",
    501: "501 Not Implemented",
    502: "502 Bad Gateway",
    503: "503 Service Unavailable",
    504: "504 Gateway Timeout",
    505: "505 Http Version Not Supported",
}

HTTP_HEADER_CACHE_CONTROL_DEFAULT = ("Cache-Control", "private")
```

Content Type and Encoding

You instantiate `HTTPResponse` and initialize it with `content_type` and `encoding`:

```
>>> r = HTTPResponse()
>>> r.headers
[('Content-Type', 'text/html; charset=UTF-8')]
>>> r = HTTPResponse(content_type='image/gif')
>>> r.headers
[('Content-Type', 'image/gif')]

>>> r = HTTPResponse(content_type='text/plain; charset=iso-8859-4',
...                  encoding='iso-8859-4')
>>> r.headers
[('Content-Type', 'text/plain; charset=iso-8859-4')]
```

Buffered Output

`HTTPResponse` has two methods to buffer output: `write` and `write_bytes`.

Method `write` let you buffer response before it actually being passed to application server. The `write` method does encoding of input chunk to bytes accordingly to response encoding.

Method `write_bytes` buffers output bytes.

Other Members

Here are some attributes available in `HTTPResponse`:

- `cache` - setup `HTTPCachePolicy`. Defaults to `private` cache policy.
- `cache_dependency` - a list of keys; used to setup dependency for given request thus effectively invalidating cached response depending on some application logic. It is a hook for integration with wheezy.caching.
- `headers` - list of headers to be returned to browser; the header must be a tuple of two: `(name, value)`. No checks for duplicates.
- `cookies` - list of cookies to set in response. This list contains `HTTPCookie` objects.

Redirect Responses

There are a number of handy preset redirect responses:

- `permanent_redirect()` - returns permanent redirect response. The HTTP response status `301 Moved Permanently` is used for permanent redirection.
- `redirect()`, `found()` - returns redirect response. The HTTP response status `302 Found` is a common way of performing a redirection.
- `see_other()` - returns see other redirect response. The HTTP response status `303 See Other` is the correct manner in which to redirect web applications to a new URI, particularly after an HTTP POST has been performed. This response indicates that the correct response can be found under a different URI and should be retrieved using a GET method. The specified URI is not a substitute reference for the original resource.
- `temporary_redirect()` - returns temporary redirect response. In this occasion, the request should be repeated with another URI, but future requests can still use the original URI. In contrast to 303, the request method should not be changed when reissuing the original request. For instance, a POST request must be repeated using another POST request.

AJAX Redirect

Browsers incorrectly handle redirect response to AJAX requests, so there is used status code 207 that javascript is capable to receive and process browser redirect.

- `ajax_redirect()` - returns ajax redirect response.

Here is an example for jQuery:

```
$.ajax({
    // ...
    success: function(data, textStatus, jqXHR) {
        if (jqXHR.status == 207) {
            window.location.replace(
                jqXHR.getResponseHeader('Location'));
        } else {
            // ...
        }
    }
});
```

If AJAX response status code is 207, browser navigates to URL specified in HTTP response header `Location`.

Error Responses

There are a number of handy preset client error responses:

- `bad_request()`, `error400()` - the request cannot be fulfilled due to bad syntax.
- `unauthorized()`, `error401()` - similar to `403 Forbidden`, but specifically for use when authentication is possible but has failed or not yet been provided.
- `forbidden()`, `error402()` - The request was a legal request, but the server is refusing to respond to it.
- `not_found()`, `error404()` - The requested resource could not be found but may be available again in the future. Subsequent requests by the client are permissible.
- `method_not_allowed()`, `error405()` - a request was made of a resource using a request method not supported by that resource; for example, using GET on a form which requires data to be presented via POST, or using PUT on a read-only resource.
- `internal_error()`, `error500()` - returns internal error response.
- `http_error()` - returns a response with given status code (between 400 and 505).

JSON

There is integration with wheezy.core package in json object encoding.

- `json_response()` - returns json response. Accepts two arguments `obj` and optional `encoding` that defaults to *UTF-8*.

Here is simple example:

```
from wheezy.http import bad_request
from wheezy.http import json_response

def now_handler(request):
    if not request.ajax:
        return bad_request()
    return json_response({'now': datetime.now()})
```

Requests other than AJAX are rejected, return JSON response with current time of server.

2.3.7 Cookies

`HTTPCookie` is implemented according to rfc2109. Here is a typical usage:

```
response.cookies.append(HTTPCookie('a', value='123', options=options))
```

In case you would like delete a certain cookie:

```
response.cookies.append(HTTPCookie.delete('a', options=options))
```

Security

While the idea behind secure cookies is to protect value (via some sort of encryption, hashing, etc), this task is out of scope of this package. However you can use `Ticket` from wheezy.security package for this purpose; it supports encryption, hashing, expiration and verification.

2.3.8 Transforms

Transforms is a way to manipulate handler response accordingly to some algorithm. Typical use case includes: runtime minification, hardening readability, gzip, etc. While middleware is applied to whole application, transform in contrast to particular handler only.

Transform is any callable of this form:

```
def transform(request, response):
    return response
```

There is a general decorator capable of applying several transforms to a response. You can use it in the following way:

```
from wheezy.http.transforms import gzip_transform
from wheezy.http.transforms import response_transforms

@response_transforms(gzip_transform(compress_level=9))
def handler(request):
    return response
```

If you need apply several transforms to handler here is how you can do that:

```
@response_transforms(a_transform, b_transform)
def handler(request):
    return response
```

Order in which transforms are applied are from first argument to last:

```
a_transform => b_transform
```

GZip Transform

It is not always effective to apply gzip encoding to whole applications. While in most cases WSGI applications are deployed behind reverse proxy web server, it is more effective to use its capabilities of response compression (10-20% productivity gain with nginx). ON the other side, gzipped responses stored in cache are even better, since compression is done once before being added to cache. This is why there is a gzip transform.

Here is a definition:

```
def gzip_transform(compress_level=6, min_length=1024, vary=False):
```

`compress_level` - the compression level, between 1 and 9, where 1 is the least compression (fastest) and 9 is the most (slowest)

`min_length` - sets the minimum length, in bytes, of the first chunk in response that will be compressed. Responses shorter than this byte-length will not be compressed.

`vary` - enables response header "Vary: Accept-Encoding".

2.3.9 Cache Policy

`HTTPCachePolicy` controls cache specific http headers: Cache-Control, Pragma, Expires, Last-Modified, ETag, Vary.

Cacheability Options

While particular set of valid HTTP cache headers depends on certain use case, there are distinguished three of them:

- `no-cache` - indicates cached information should not be used and instead requests should be forwarded to the origin server.
- `private` - response is cacheable only on the client and not by shared (proxy server) caches.
- `public` - response is cacheable by clients and shared (proxy) caches.

Useful Methods

`HTTPCachePolicy` includes the following useful methods:

- `private(*fields)` - indicates that part of the response message is intended for a single user and MUST NOT be cached by a shared cache. Only valid for `public` cacheability.
- `no_cache(*fields)` - the specified field-name(s) MUST NOT be sent in the response to a subsequent request without successful re-validation with the origin server. Not valid for `no-cache` cacheability.
- `no_store()` - the purpose of the no-store directive is to prevent the inadvertent release or retention of sensitive information.
- `must_revalidate()` - because a cache MAY be configured to ignore a server's specified expiration time, and because a client request MAY include a max-stale directive (which has a similar effect), the protocol also includes a mechanism for the origin server to require re-validation of a cache entry on any subsequent use.
- `proxy_revalidate()` - the proxy-revalidate directive has the same meaning as the must-revalidate directive, except that it does not apply to non-shared user agent caches.
- `no_transform()` - the cache or proxy MUST NOT change any aspect of the entity-body that is specified by this header, including the value of the entity-body itself.
- `append_extension(extension)` - appends the `extension` to the Cache-Control HTTP header.
- `max_age(delta)` - accept a response whose age is no greater than the specified time in seconds. Not valid for `no-cache` cacheability.
- `smax_age(delta)` - if a response includes an s-maxage directive, then for a shared cache (but not for a private cache). Not valid for `no-cache` cacheability.
- `expires(when)` - gives the date/time after which the response is considered stale. Not valid for `no-cache` cacheability.
- `last_modified(when)` - the Last-Modified entity-header field indicates the date and time at which the origin server believes the variant was last modified. Not valid for `no-cache` cacheability.
- `etag(tag)` - provides the current value of the entity tag for the requested variant. Not valid for `no-cache` cacheability.
- `vary(*headers)` - indicates the set of request-header fields that fully determines, while the response is fresh, whether a cache is permitted to use the response to reply to a subsequent request without re-validation. Not valid for `no-cache` cacheability.

Examples

You can use `extend(headers)` method to update `headers` with this cache policy (this is what `HTTPResponse` does when `cache` attribute is set):

```
>>> headers = []
>>> p = HTTPCachePolicy('no-cache')
>>> p.extend(headers)
>>> headers
[('Cache-Control', 'no-cache'),
('Pragma', 'no-cache'),
('Expires', '-1')]
```

Public caching headers:

```
>>> from datetime import datetime, timedelta
>>> from wheezy.core.datetime import UTC
>>> when = datetime(2011, 9, 20, 15, 00, tzinfo=UTC)
>>> headers = []
>>> p = HTTPCachePolicy('public')
>>> p.last_modified(when)
>>> p.expires(when + timedelta(hours=1))
>>> p.etag('abc')
>>> p.vary()
>>> p.extend(headers)
>>> headers # doctest: +NORMALIZE_WHITESPACE
[('Cache-Control', 'public'),
('Expires', 'Tue, 20 Sep 2011 16:00:00 GMT'),
('Last-Modified', 'Tue, 20 Sep 2011 15:00:00 GMT'),
('ETag', 'abc'),
('Vary', '*')]
```

While you do not directly make a call to extend headers from cache policy, it is still useful to experiment within a python console.

2.3.10 Cache Profile

`CacheProfile` combines a number of settings applicable to http cache policy as well as server side cache.

Cache Location

`CacheProfile` supports the following list of valid cache locations:

- `none` - no server or client cache.
- `server` - only server side caching, no client cache.
- `client` - only client side caching, no server cache.
- `both` - server and client caching.
- `public` - server and client caching including intermediate proxies.

Here is a map between cache profile cacheability and http cache policy:

```
    "server": "no-cache",
    "client": "private",
    "both": "private",  # server and client
    "public": "public",
}
```

Cache profile method `cache_policy` is adapted according the above map.

Typical Use

You create a cache profile by instantiating `CacheProfile` and passing in the following arguments:

- `location` - must fall into one of acceptable values as defined by `SUPPORTED`.
- `duration` - time for the cache item to be cached.
- `no_store` - instructs state of `no-store` http response header.
- `vary_query` - a list of query items that should be included into cache key.
- `vary_form` - a list of form items that should be included into cache key.
- `vary_environ` - a list of environ items that should be included into cache key (particularly useful to vary by HTTP headers, request scheme, etc).
- `vary_cookies` - a list of cookies that should be included into cache key.
- `http_vary` - manages HTTP cache policy *Very* header.
- `etag_func` - a function used to setup HTTP cache policy ETag header. See `make_etag()` and `make_etag_crc32()`.
- `namespace` - a namespace to be used in server cache operations.
- `enabled` - determines whenever this cache profile is enabled.

Here is an example:

```
cache_profile = CacheProfile('client', duration=timedelta(minutes=15))

cache_profile = CacheProfile('both', duration=15)
```

It is recommended to define cache profiles in a separate module and import them as needed into a various parts of application. This way you can achieve better control with a single place of change.

2.3.11 Content Cache

Content caching is the most effective type of cache. This way your application code doesn't have to process to determine a valid response to user. Instead a response is returned from cache. Since there is no heavy processing and just simple operation to get an item from cache, it should be super fast. However not every request can be cached and whether it can completely depends on your application.

If you show a list of goods and it has not changed in any way (price is the same, etc.) why would you make several calls per second every time it requested and regenerate the page again? You can apply cache profile to response and it will be cached according to it rules.

What happens if the price has been changed, but the list of goods cacheability was set to 15 mins? How to invalidate the cache? This is where `CacheDependency` comes to the rescue. The core feature of cache dependency is implemented in package wheezy.caching, however http module supports its integration.

Cache Contract

Cache contract requires: `get(key, namespace)`, `set(key, value, time, namespace)`, `set_multi(mapping, time, namespace)` and `incr(self, key, delta=1, namespace=None, initial_value=None)`. Look at wheezy.caching package for more details.

@response_cache

`response_cache()` decorator is used to apply cache feature to handler. Here is an example that includes also `CacheDependency`:

```
from wheezy.caching.patterns import Cached
from wheezy.http import CacheProfile
from wheezy.http import none_cache_profile
from wheezy.http import response_cache
from myapp import cache

cached = Cached(cache, time=15)
cache_profile = CacheProfile('server', duration=15)

@response_cache(cache_profile)
def list_of_goods(request):
    ...
    response.cache_dependency.append('list_of_goods:%s:' % catalog_id)
    return response

@response_cache(none_cache_profile)
def change_price(request):
    ...
    cached.dependency.delete('list_of_goods:%s:' % catalog_id)
    return response
```

While `list_of_goods` is being cached, `change_price` handler effectively invalidates `list_of_goods` cache result, so next call will fetch an updated list.

Note, cache dependency keys must not end with a number.

Cache Middleware

The `response_cache()` decorator is applied to handler. It is pretty far from the WSGI entry point, there are number of middlewares as well as routing in between (all these are relatively time consuming, especially routing). What if we were able determine cache profile for the given request earlier, being the first middleware in the chain. This is where `HTTPCacheMiddleware` comes to the scene.

`HTTPCacheMiddleware` serves exactly this purpose. It is initialized with two arguments:

- `cache` - a cache to be used (must be thread safe, see wheezy.caching for various implementations).
- `middleware_vary` - a strategy to be used to determine cache profile key for the incoming request.

Here is an example:

```
cache = ...
options = {
    'http_cache': cache
}

main = WSGIApplication([
    http_cache_middleware_factory()
], options)
```

`middleware_vary` is an instance of `RequestVary`. By default it varies cache key by HTTP method and path. Let assume we would like vary middleware key by HTTP scheme:

```
options = {
    ...
    'http_cache_middleware_vary': RequestVary(
        environ=['wsgi.url_scheme'])
}
```

Request Vary

`RequestVary` is designed to compose a key depending on number of values, including: headers, query, form and environ. It always varies by request method and path.

Here is a list of arguments that can be passed during initialization:

- `query` - a list of request url query items.
- `form` - a list of form items submitted via http POST method.
- `environ` - a list of items from environ.

The following example will vary incoming request by request url query parameter *q*:

```
request_vary = RequestVary(query=['q'])
```

Note that you can vary by HTTP headers via environ names. A missing value is distinguished from an empty one.

`RequestVary` is used by `CacheProfile` and `HTTPCacheMiddleware` internally.

2.3.12 WSGI Adapters

wheezy.http providers middleware adapters to be used for integration with other WSGI applications:

- `WSGIAdapterMiddleware` - adapts WSGI application response (initialization requires `wsgi_app` argument to be passed).
- `EnvironCacheAdapterMiddleware` - adapts WSGI environ variables: `wheezy.http.cache_policy`, `wheezy.http.cache_profile`, `wheezy.http.cache_dependency` for http content caching middleware.

See the demo example in the wsgi_adapter application.

2.4 Functional Testing

Functional testing is a type of black box testing. Functions are tested by feeding them input and examining the output. Internal program structure is rarely considered.

Let take a look at functional tests for *Hello World* application:

```
from wheezy.http.functional import WSGIClient


class HelloWorldTestCase(unittest.TestCase):
    def setUp(self):
        from helloworld import main
```

(continues on next page)

(continued from previous page)

```
        self.client = WSGIClient(main)

    def tearDown(self):
        del self.client
        self.client = None

    def test_welcome(self):
        """Ensure welcome page is rendered."""
        assert 200 == self.client.get("/")
        assert "Hello" in self.client.content

    def test_not_found(self):
        """Ensure not found status code."""
        assert 404 == self.client.get("/x")
```

wheezy.http comes with a `WSGIClient` that simulates calls to a WSGI application.

While developing functional tests it is recommended to distinguish three primary actors:

- Page
- Functional Mixin
- Test Case

Let's demo this idea in a scenario where we would like to test a signin process.

2.4.1 Page

Page provides a number of asserts to prove the current content is related to given page. Since this page will be used to submit signin information we need find a form as well. Here is our signin page:

```
class SignInPage(object):

    def __init__(self, client):
        assert '- Sign In</title>' in client.content
        assert AUTH_COOKIE not in client.cookies
        assert XSRF_NAME in client.cookies
        self.client = client
        self.form = client.form

    def signin(self, username, password):
        form = self.form
        form.username = username
        form.password = password
        self.client.submit(form)
        return self.client.form.errors()
```

We add as much asserts as necessary to prove this is the signin page. We look at title, check cookies and select form. `signin` method implements a simple use case to initialize a form with parameters passed, submit the form and return back any errors found.

Consider using `PageMixin` to simplify form submit use cases.

2.4.2 Functional Mixin

Functional mixin is more like a high level actor. While considered to be developed as mixin, your actual test case can combine them as much as necessary, to fulfill its goal. Here is a singin mixin:

```
class SignInMixin(object):

    def signin(self, username, password):
        client = self.client
        assert 200 == client.get('/en/signin')
        page = SignInPage(client)
        return page.signin(username, password)
```

It is up to functional mixin to implement a particular use case. However it is recommended that its method represents an operation particular to given domain, abstracting details like url, form, etc.

2.4.3 Test Case

While page and functional mixin play distinct simple roles, test case tries to get as much as possible to accomplish a number of use cases. Here is a test case:

```
class SignInTestCase(unittest.TestCase, SignInMixin):

    def setUp(self):
        self.client = WSGIClient(main)

    def tearDown(self):
        del self.client
        self.client = None

    def test_validation_error(self):
        """ Ensure sigin page displays field validation errors.
        """
        errors = self.signin('', '')
        assert 2 == len(errors)
        assert AUTH_COOKIE not in self.client.cookies

    def test_valid_user(self):
        """ Ensure sigin is successful.
        """
        self.signin('demo', 'P@ssw0rd')
        assert 200 == self.client.follow()
        assert AUTH_COOKIE in self.client.cookies
        assert XSRF_NAME not in self.client.cookies
        assert 'Welcome <b>demo' in self.client.content
```

Test case can use many functional mixins to accomplish its goal. Test case in general is a set of conditions under which we can determine whether an application is working correctly or not. The mechanism for determining whether a software program has passed or failed such a test is known as a test oracle. In some settings, an oracle could be a requirement or use case, while in others it could be a heuristic. It may take many test cases to determine that a software program or system is considered sufficiently scrutinized to be released. Being able combine and reuse test case building blocks is crucial.

2.4.4 Benchmark

You can benchmark your test cases with `wheezy.core.benchmark.Benchmark`. Here is an example:

```
""" ``benchmark_views`` module.
"""

from wheezy.core.benchmark import Benchmark

from public.web.tests.test_views import PublicTestCase


class BenchmarkTestCase(PublicTestCase):

    def runTest(self):
        """ Perform bachmark and print results.
        """
        p = Benchmark((
            self.test_home,
            self.test_about,
            self.test_static_files,
            self.test_static_file_not_found,
            self.test_static_file_forbidden,
            self.test_static_file_gzip,
            self.test_head_static_file
            ), 1000)
        p.report('public', baselines={
                 'test_home': 1.0,
                 'test_about': 0.926,
                 'test_static_files': 1.655,
                 'test_static_file_not_found': 0.64,
                 'test_static_file_forbidden': 0.62,
                 'test_static_file_gzip': 8.91,
                 'test_head_static_file': 9.08
        })
```

Report

Sample output:

```
public: 7 x 1000
baseline throughput change target
  100.0%     839rps  +0.0% test_home
   96.2%     807rps  +3.9% test_about
  235.7%    1979rps +42.4% test_static_files
   72.4%     608rps +13.1% test_static_file_not_found
   72.3%     607rps +16.6% test_static_file_forbidden
 1141.4%    9585rps +28.1% test_static_file_gzip
 1193.6%   10023rps +31.5% test_head_static_file
```

Each of seven test cases has been run 1000 times. It shows productivity gain from first test case (it serves as a baseline for others), throughput in requests per second, change from `baselines` argument passed to `report` method and targeted being benchmarked.

Report is being printed as results become available.

Consider using `BenchmakrMixin` to get benchmark results close to WSGI application entry point.

Organizing Benchmarks

It is recommended keep benchmark tests separated from others tests in files with prefix `benchmark`, e.g. `benchmark_views.py`. This way they can be run separately. Here is an example how to run only the benchmark tests with `nose`:

```
$ nosetests-2.7 -qs -m benchmark src/
```

This method of benchmarking does not involve the web server layer, nor http traffic, instead it gives you an idea of how performance of your handlers evolves over time.

Profiling

Since benchmark does certain workload on your application that workload is a good start point for profiling your code as well as analyzing productivity bottlenecks.

Here we are running the profiler:

```
$ nosetests-2.7 -qs -m benchmark --with-profile \
            --profile-stats-file=profile.pstats src/
```

Profiling results can be further analyzed with:

```
gprof2dot.py -f pstats profile.pstats | dot -Tpng -o profile.png
```

Profiling your application lets you determine performance critical places that might require further optimization.

Performance

You can boost `WSGIClient` form parsing performance by installing the lxml package. `WSGIClient` tries to use `HTMLParser` from the `lxml.etree` package and if it is not available falls back to the default parser in the standard library.

2.5 Modules

2.5.1 wheezy.http

2.5.2 wheezy.http.application

2.5.3 wheezy.http.authorization

2.5.4 wheezy.http.cache

2.5.5 wheezy.http.cachepolicy

2.5.6 wheezy.http.cacheprofile

2.5.7 wheezy.http.config

```
def bootstrap_http_defaults(options):
    """Bootstraps http default options."""
    options.setdefault("ENCODING", "UTF-8")
    options.setdefault("MAX_CONTENT_LENGTH", 4 * 1024 * 1024)
    options.setdefault("HTTP_COOKIE_DOMAIN", None)
    options.setdefault("HTTP_COOKIE_SAMESITE", None)
    options.setdefault("HTTP_COOKIE_SECURE", False)
    options.setdefault("HTTP_COOKIE_HTTPONLY", False)
    return None
```

2.5.8 wheezy.http.cookie

2.5.9 wheezy.http.functional

2.5.10 wheezy.http.method

2.5.11 wheezy.http.middleware

2.5.12 wheezy.http.parse

2.5.13 wheezy.http.request

2.5.14 wheezy.http.response

2.5.15 wheezy.http.transforms