WebOb Documentation

Release 1.5.1

Ian Bicking and contributors

January 29, 2016

Contents

WebOb provides objects for HTTP requests and responses. Specifically it does this by wrapping the WSGI request environment and response status/headers/app_iter(body).

The request and response objects provide many conveniences for parsing HTTP request and forming HTTP responses. Both objects are read/write: as a result, WebOb is also a nice way to create HTTP requests and parse HTTP responses; however, we won't cover that use case in this document. The reference documentation shows many examples of creating requests.

CHAPTER 1

WebOb Reference

Contents

1.1 Introduction

This document covers all the details of the Request and Response objects. It is written to be testable with doctest – this affects the flavor of the documentation, perhaps to its detriment. But it also means you can feel confident that the documentation is correct.

This is a somewhat different approach to reference documentation compared to the extracted documentation for the `request` and `response`.

1.2 Request

The primary object in WebOb is `webob.Request`, a wrapper around a WSGI environment.

The basic way you create a request object is simple enough:

```
>>> from webob import Request
>>> environ = {'wsgi.url_scheme': 'http', ...}
>>> req = Request(environ)
```

(Note that the WSGI environment is a dictionary with a dozen required keys, so it's a bit lengthly to show a complete example of what it would look like – usually your WSGI server will create it.)

The request object *wraps* the environment; it has very little internal state of its own. Instead attributes you access read and write to the environment dictionary.

You don't have to understand the details of WSGI to use this library; this library handles those details for you. You also don't have to use this exclusively of other libraries. If those other libraries also keep their state in the environment, multiple wrappers can coexist. Examples of libraries that can coexist include paste.wsgiwrappers.Request (used by Pylons) and yaro.Request.

The WSGI environment has a number of required variables. To make it easier to test and play around with, the `Request` class has a constructor that will fill in a minimal environment:

```
>>> req = Request.blank('/article?id=1')
>>> from pprint import pprint
>>> pprint(req.environ)
{'HTTP_HOST': 'localhost:80',
 'PATH_INFO': '/article',
 'QUERY_STRING': 'id=1',
 'REQUEST_METHOD': 'GET',
 'SCRIPT_NAME': '',
 'SERVER_NAME': 'localhost',
 'SERVER_PORT': '80',
 'SERVER_PROTOCOL': 'HTTP/1.0',
 'wsgi.errors': <open file '<stderr>', mode 'w' at ...>,
 'wsgi.input': <...IO... object at ...>,
 'wsgi.multiprocess': False,
 'wsgi.multithread': False,
 'wsgi.run_once': False,
 'wsgi.url_scheme': 'http',
 'wsgi.version': (1, 0)}
```

1.2.1 Request Body

`req.body` is a file-like object that gives the body of the request (e.g., a POST form, the body of a PUT, etc). It's kind of boring to start, but you can set it to a string and that will be turned into a file-like object. You can read the entire body with *`req.body`*.

```
>>> hasattr(req.body_file, 'read')
True
>>> req.body
''
>>> req.method = 'PUT'
>>> req.body = 'test'
>>> hasattr(req.body_file, 'read')
True
```

```
>>> req.body
'test'
```

1.2.2 Method & URL

All the normal parts of a request are also accessible through the request object:

```
>>> req.method
'PUT'
>>> req.scheme
'http'
>>> req.script_name  # The base of the URL
''
>>> req.script_name = '/blog' # make it more interesting
>>> req.path_info    # The yet-to-be-consumed part of the URL
'/article'
>>> req.content_type # Content-Type of the request body
''
>>> print req.remote_user  # The authenticated user (there is none set)
None
>>> print req.remote_addr  # The remote IP
None
>>> req.host
'localhost:80'
>>> req.host_url
'http://localhost'
>>> req.application_url
'http://localhost/blog'
>>> req.path_url
'http://localhost/blog/article'
>>> req.url
'http://localhost/blog/article?id=1'
>>> req.path
'/blog/article'
>>> req.path_qs
'/blog/article?id=1'
>>> req.query_string
'id=1'
```

You can make new URLs:

```
>>> req.relative_url('archive')
'http://localhost/blog/archive'
```

For parsing the URLs, it is often useful to deal with just the next path segment on PATH_INFO:

```
>>> req.path_info_peek() # Doesn't change request
'article'
>>> req.path_info_pop()  # Does change request!
'article'
>>> req.script_name
'/blog/article'
>>> req.path_info
''
```

1.2.3 Headers

All request headers are available through a dictionary-like object *req.headers*. Keys are case-insensitive.

```
>>> req.headers['Content-Type'] = 'application/x-www-urlencoded'
>>> sorted(req.headers.items())
[('Content-Length', '4'), ('Content-Type', 'application/x-www-urlencoded'), ('Host', 'localhost:80')]
>>> req.environ['CONTENT_TYPE']
'application/x-www-urlencoded'
```

1.2.4 Query & POST variables

Requests can have variables in one of two locations: the query string (`?id=1`), or in the body of the request (generally a POST form). Note that even POST requests can have a query string, so both kinds of variables can exist at the same time. Also, a variable can show up more than once, as in `?check=a&check=b`.

For these variables WebOb uses a *MultiDict*, which is basically a dictionary wrapper on a list of key/value pairs. It looks like a single-valued dictionary, but you can access all the values of a key with *.getall(key)* (which always returns a list, possibly an empty list). You also get all key/value pairs when using `.items()` and all values with `.values()`.

Some examples:

```
>>> req = Request.blank('/test?check=a&check=b&name=Bob')
>>> req.GET
MultiDict([(u'check', u'a'), (u'check', u'b'), (u'name', u'Bob')])
>>> req.GET['check']
u'b'
>>> req.GET.getall('check')
[u'a', u'b']
>>> req.GET.items()
[(u'check', u'a'), (u'check', u'b'), (u'name', u'Bob')]
```

We'll have to create a request body and change the method to get `POST`. Until we do that, the variables are boring:

```
>>> req.POST
<NoVars: Not a form request>
>>> req.POST.items()  # NoVars can be read like a dict, but not written
[]
>>> req.method = 'POST'
>>> req.body = 'name=Joe&email=joe@example.com'
>>> req.POST
MultiDict([(u'name', u'Joe'), (u'email', u'joe@example.com')])
>>> req.POST['name']
u'Joe'
```

Often you won't care where the variables come from. (Even if you care about the method, the location of the variables might not be important.) There is a dictionary called *req.params* that contains variables from both sources:

```
>>> req.params
NestedMultiDict([(u'check', u'a'), (u'check', u'b'), (u'name', u'Bob'), (u'name', u'Joe'), (u'email',
>>> req.params['name']
u'Bob'
>>> req.params.getall('name')
[u'Bob', u'Joe']
>>> for name, value in req.params.items():
...     print '%s: %r' % (name, value)
check: u'a'
```

```
check: u'b'
name: u'Bob'
name: u'Joe'
email: u'joe@example.com'
```

The `POST` and `GET` nomenclature is historical – *`req.GET`* can be used for non-GET requests to access query parameters, and *`req.POST`* can also be used for PUT requests with the appropriate Content-Type.

```
>>> req = Request.blank('/test?check=a&check=b&name=Bob')
>>> req.method = 'PUT'
>>> req.body = body = 'var1=value1&var2=value2&rep=1&rep=2'
>>> req.environ['CONTENT_LENGTH'] = str(len(req.body))
>>> req.environ['CONTENT_TYPE'] = 'application/x-www-form-urlencoded'
>>> req.GET
MultiDict([(u'check', u'a'), (u'check', u'b'), (u'name', u'Bob')])
>>> req.POST
MultiDict([(u'var1', u'value1'), (u'var2', u'value2'), (u'rep', u'1'), (u'rep', u'2')])
```

Unicode Variables

Submissions are non-unicode (`str`) strings, unless some character set is indicated. A client can indicate the character set with `Content-Type: application/x-www-form-urlencoded; charset=utf8`, but very few clients actually do this (sometimes XMLHttpRequest requests will do this, as JSON is always UTF8 even when a page is served with a different character set). You can force a charset, which will affect all the variables:

```
>>> req.charset = 'utf8'
>>> req.GET
MultiDict([(u'check', u'a'), (u'check', u'b'), (u'name', u'Bob')])
```

1.2.5 Cookies

Cookies are presented in a simple dictionary. Like other variables, they will be decoded into Unicode strings if you set the charset.

```
>>> req.headers['Cookie'] = 'test=value'
>>> req.cookies
MultiDict([(u'test', u'value')])
```

1.2.6 Modifying the request

The headers are all modifiable, as are other environmental variables (like *`req.remote_user`*, which maps to `request.environ['REMOTE_USER']`).

If you want to copy the request you can use *`req.copy()`*; this copies the `environ` dictionary, and the request body from `environ['wsgi.input']`.

The method *`req.remove_conditional_headers(remove_encoding=True)`* can be used to remove headers that might result in a `304 Not Modified` response. If you are writing some intermediary it can be useful to avoid these headers. Also if `remove_encoding` is true (the default) then any `Accept-Encoding` header will be removed, which can result in gzipped responses.

1.2.7 Header Getters

In addition to *req.headers*, there are attributes for most of the request headers defined by the HTTP 1.1 specification. These attributes often return parsed forms of the headers.

Accept-* headers

There are several request headers that tell the server what the client accepts. These are `accept` (the Content-Type that is accepted), `accept_charset` (the charset accepted), `accept_encoding` (the Content-Encoding, like gzip, that is accepted), and `accept_language` (generally the preferred language of the client).

The objects returned support containment to test for acceptability. E.g.:

```
>>> 'text/html' in req.accept
True
```

Because no header means anything is potentially acceptable, this is returning True. We can set it to see more interesting behavior (the example means that `text/html` is okay, but `application/xhtml+xml` is preferred):

```
>>> req.accept = 'text/html;q=0.5, application/xhtml+xml;q=1'
>>> req.accept
<MIMEAccept('text/html;q=0.5, application/xhtml+xml')>
>>> 'text/html' in req.accept
True
```

There are a few methods for different strategies of finding a match.

```
>>> req.accept.best_match(['text/html', 'application/xhtml+xml'])
'application/xhtml+xml'
```

If we just want to know everything the client prefers, in the order it is preferred:

```
>>> list(req.accept)
['application/xhtml+xml', 'text/html']
```

For languages you'll often have a "fallback" language. E.g., if there's nothing better then use `en-US` (and if `en-US` is okay, ignore any less preferrable languages):

```
>>> req.accept_language = 'es, pt-BR'
>>> req.accept_language.best_match(['en-GB', 'en-US'], default_match='en-US')
'en-US'
>>> req.accept_language.best_match(['es', 'en-US'], default_match='en-US')
'es'
```

Your fallback language must appear both in the `offers` and as the `default_match` to insure that it is returned as a best match if the client specified a preference for it.

```
>>> req.accept_language = 'en-US;q=0.5, en-GB;q=0.2'
>>> req.accept_language.best_match(['en-GB'], default_match='en-US')
'en-GB'
>>> req.accept_language.best_match(['en-GB', 'en-US'], default_match='en-US')
'en-US'
```

Conditional Requests

There a number of ways to make a conditional request. A conditional request is made when the client has a document, but it is not sure if the document is up to date. If it is not, it wants a new version. If the document is up to date then it doesn't want to waste the bandwidth, and expects a `304 Not Modified` response.

ETags are generally the best technique for these kinds of requests; this is an opaque string that indicates the identity of the object. For instance, it's common to use the mtime (last modified) of the file, plus the number of bytes, and maybe a hash of the filename (if there's a possibility that the same URL could point to two different server-side filenames based on other variables). To test if a 304 response is appropriate, you can use:

```
>>> server_token = 'opaque-token'
>>> server_token in req.if_none_match # You shouldn't return 304
False
>>> req.if_none_match = server_token
>>> req.if_none_match
<ETag opaque-token>
>>> server_token in req.if_none_match # You *should* return 304
True
```

For date-based comparisons If-Modified-Since is used:

```
>>> from webob import UTC
>>> from datetime import datetime
>>> req.if_modified_since = datetime(2006, 1, 1, 12, 0, tzinfo=UTC)
>>> req.headers['If-Modified-Since']
'Sun, 01 Jan 2006 12:00:00 GMT'
>>> server_modified = datetime(2005, 1, 1, 12, 0, tzinfo=UTC)
>>> req.if_modified_since and req.if_modified_since >= server_modified
True
```

For range requests there are two important headers, If-Range (which is form of conditional request) and Range (which requests a range). If the If-Range header fails to match then the full response (not a range) should be returned:

```
>>> req.if_range
<Empty If-Range>
>>> req.if_range.match(etag='some-etag', last_modified=datetime(2005, 1, 1, 12, 0))
True
>>> req.if_range = 'opaque-etag'
>>> req.if_range.match(etag='other-etag')
False
>>> req.if_range.match(etag='opaque-etag')
True
```

You can also pass in a response object with:

```
>>> from webob import Response
>>> res = Response(etag='opaque-etag')
>>> req.if_range.match_response(res)
True
```

To get the range information:

```
>>> req.range = 'bytes=0-100'
>>> req.range
<Range ranges=(0, 101)>
>>> cr = req.range.content_range(length=1000)
>>> cr.start, cr.stop, cr.length
(0, 101, 1000)
```

Note that the range headers use *inclusive* ranges (the last byte indexed is included), where Python always uses a range where the last index is excluded from the range. The `.stop` index is in the Python form.

Another kind of conditional request is a request (typically PUT) that includes If-Match or If-Unmodified-Since. In this case you are saying "here is an update to a resource, but don't apply it if someone else has done something since I last got the resource". If-Match means "do this if the current ETag matches the ETag I'm giving". If-Unmodified-Since means "do this if the resource has remained unchanged".

```
>>> server_token in req.if_match # No If-Match means everything is ok
True
>>> req.if_match = server_token
>>> server_token in req.if_match # Still OK
True
>>> req.if_match = 'other-token'
>>> # Not OK, should return 412 Precondition Failed:
>>> server_token in req.if_match
False
```

For more on this kind of conditional request, see Detecting the Lost Update Problem Using Unreserved Checkout.

1.2.8 Calling WSGI Applications

The request object can be used to make handy subrequests or test requests against WSGI applications. If you want to make subrequests, you should copy the request (with `req.copy()`) before sending it to multiple applications, since applications might modify the request when they are run.

There's two forms of the subrequest. The more primitive form is this:

```
>>> req = Request.blank('/')
>>> def wsgi_app(environ, start_response):
...     start_response('200 OK', [('Content-type', 'text/plain')])
...     return ['Hi!']
>>> req.call_application(wsgi_app)
('200 OK', [('Content-type', 'text/plain')], ['Hi!'])
```

Note it returns `(status_string, header_list, app_iter)`. If `app_iter.close()` exists, it is your responsibility to call it.

A handier response can be had with:

```
>>> res = req.get_response(wsgi_app)
>>> res
<Response ... 200 OK>
>>> res.status
'200 OK'
>>> res.headers
ResponseHeaders([('Content-type', 'text/plain')])
>>> res.body
'Hi!'
```

You can learn more about this response object in the *Response* section.

1.2.9 Ad-Hoc Attributes

You can assign attributes to your request objects. They will all go in `environ['webob.adhoc_attrs']` (a dictionary).

```
>>> req = Request.blank('/')
>>> req.some_attr = 'blah blah blah'
>>> new_req = Request(req.environ)
>>> new_req.some_attr
'blah blah blah'
>>> req.environ['webob.adhoc_attrs']
{'some_attr': 'blah blah blah'}
```

1.3 Response

The *webob.Response* object contains everything necessary to make a WSGI response. Instances of it are in fact WSGI applications, but it can also represent the result of calling a WSGI application (as noted in *Calling WSGI Applications*). It can also be a way of accumulating a response in your WSGI application.

A WSGI response is made up of a status (like `200 OK`), a list of headers, and a body (or iterator that will produce a body).

1.3.1 Core Attributes

The core attributes are unsurprising:

```
>>> from webob import Response
>>> res = Response()
>>> res.status
'200 OK'
>>> res.headerlist
[('Content-Type', 'text/html; charset=UTF-8'), ('Content-Length', '0')]
>>> res.body
''
```

You can set any of these attributes, e.g.:

```
>>> res.status = 404
>>> res.status
'404 Not Found'
>>> res.status_code
404
>>> res.headerlist = [('Content-type', 'text/html')]
>>> res.body = 'test'
>>> print res
404 Not Found
Content-type: text/html
Content-Length: 4

test
>>> res.body = u"test"
Traceback (most recent call last):
  ...
TypeError: You cannot set Response.body to a unicode object (use Response.text)
>>> res.text = u"test"
Traceback (most recent call last):
  ...
AttributeError: You cannot access Response.text unless charset is set
>>> res.charset = 'utf8'
>>> res.text = u"test"
>>> res.body
'test'
```

You can set any attribute with the constructor, like `Response(charset='utf8')`

1.3.2 Headers

In addition to `res.headerlist`, there is dictionary-like view on the list in `res.headers`:

```
>>> res.headers
ResponseHeaders([('Content-Type', 'text/html; charset=utf8'), ('Content-Length', '4')])
```

This is case-insensitive. It can support multiple values for a key, though only if you use `res.headers.add(key, value)` or read them with `res.headers.getall(key)`.

1.3.3 Body & app_iter

The `res.body` attribute represents the entire body of the request as a single string (not unicode, though you can set it to unicode if you have a charset defined). There is also a `res.app_iter` attribute that reprsents the body as an iterator. WSGI applications return these `app_iter` iterators instead of strings, and sometimes it can be problematic to load the entire iterator at once (for instance, if it returns the contents of a very large file). Generally it is not a problem, and often the iterator is something simple like a one-item list containing a string with the entire body.

If you set the body then Content-Length will also be set, and an `res.app_iter` will be created for you. If you set `res.app_iter` then Content-Length will be cleared, but it won't be set for you.

There is also a file-like object you can access, which will update the app_iter in-place (turning the app_iter into a list if necessary):

```
>>> res = Response(content_type='text/plain', charset=None)
>>> f = res.body_file
>>> f.write('hey')
>>> f.write(u'test')
Traceback (most recent call last):
  . . .
TypeError: You can only write unicode to Response if charset has been set
>>> f.encoding
>>> res.charset = 'utf8'
>>> f.encoding
'utf8'
>>> f.write(u'test')
>>> res.app_iter
['', 'hey', 'test']
>>> res.body
'heytest'
```

1.3.4 Header Getters

Like Request, HTTP response headers are also available as individual properties. These represent parsed forms of the headers.

Content-Type is a special case, as the type and the charset are handled through two separate properties:

```
>>> res = Response()
>>> res.content_type = 'text/html'
>>> res.charset = 'utf8'
>>> res.content_type
'text/html'
>>> res.headers['content-type']
'text/html; charset=utf8'
>>> res.content_type = 'application/atom+xml'
>>> res.content_type_params
{'charset': 'utf8'}
>>> res.content_type_params = {'type': 'entry', 'charset': 'utf8'}
```

```
>>> res.headers['content-type']
'application/atom+xml; charset=utf8; type=entry'
```

Other headers:

```
>>> # Used with a redirect:
>>> res.location = 'http://localhost/foo'

>>> # Indicates that the server accepts Range requests:
>>> res.accept_ranges = 'bytes'

>>> # Used by caching proxies to tell the client how old the
>>> # response is:
>>> res.age = 120

>>> # Show what methods the client can do; typically used in
>>> # a 405 Method Not Allowed response:
>>> res.allow = ['GET', 'PUT']

>>> # Set the cache-control header:
>>> res.cache_control.max_age = 360
>>> res.cache_control.no_transform = True

>>> # Tell the browser to treat the response as an attachment:
>>> res.content_disposition = 'attachment; filename=foo.xml'

>>> # Used if you had gzipped the body:
>>> res.content_encoding = 'gzip'

>>> # What language(s) are in the content:
>>> res.content_language = ['en']

>>> # Seldom used header that tells the client where the content
>>> # is from:
>>> res.content_location = 'http://localhost/foo'

>>> # Seldom used header that gives a hash of the body:
>>> res.content_md5 = 'big-hash'

>>> # Means we are serving bytes 0-500 inclusive, out of 1000 bytes total:
>>> # you can also use the range setter shown earlier
>>> res.content_range = (0, 501, 1000)

>>> # The length of the content; set automatically if you set
>>> # res.body:
>>> res.content_length = 4

>>> # Used to indicate the current date as the server understands
>>> # it:
>>> res.date = datetime.now()

>>> # The etag:
>>> res.etag = 'opaque-token'
>>> # You can generate it from the body too:
>>> res.md5_etag()
>>> res.etag
'1B2M2Y8AsgTpgAmY7PhCfg'
```

```
>>> # When this page should expire from a cache (Cache-Control
>>> # often works better):
>>> import time
>>> res.expires = time.time() + 60*60 # 1 hour

>>> # When this was last modified, of course:
>>> res.last_modified = datetime(2007, 1, 1, 12, 0, tzinfo=UTC)

>>> # Used with 503 Service Unavailable to hint the client when to
>>> # try again:
>>> res.retry_after = 160

>>> # Indicate the server software:
>>> res.server = 'WebOb/1.0'

>>> # Give a list of headers that the cache should vary on:
>>> res.vary = ['Cookie']
```

Note in each case you can general set the header to a string to avoid any parsing, and set it to None to remove the header (or do something like `del res.vary`).

In the case of date-related headers you can set the value to a `datetime` instance (ideally with a UTC timezone), a time tuple, an integer timestamp, or a properly-formatted string.

After setting all these headers, here's the result:

```
>>> for name, value in res.headerlist:
...     print '%s: %s' % (name, value)
Content-Type: application/atom+xml; charset=utf8; type=entry
Location: http://localhost/foo
Accept-Ranges: bytes
Age: 120
Allow: GET, PUT
Cache-Control: max-age=360, no-transform
Content-Disposition: attachment; filename=foo.xml
Content-Encoding: gzip
Content-Language: en
Content-Location: http://localhost/foo
Content-MD5: big-hash
Content-Range: bytes 0-500/1000
Content-Length: 4
Date: ... GMT
ETag: ...
Expires: ... GMT
Last-Modified: Mon, 01 Jan 2007 12:00:00 GMT
Retry-After: 160
Server: WebOb/1.0
Vary: Cookie
```

You can also set Cache-Control related attributes with `req.cache_expires(seconds, **attrs)`, like:

```
>>> res = Response()
>>> res.cache_expires(10)
>>> res.headers['Cache-Control']
'max-age=10'
>>> res.cache_expires(0)
>>> res.headers['Cache-Control']
'max-age=0, must-revalidate, no-cache, no-store'
>>> res.headers['Expires']
```

```
'... GMT'
```

You can also use the timedelta constants defined, e.g.:

```
>>> from webob import *
>>> res = Response()
>>> res.cache_expires(2*day+4*hour)
>>> res.headers['Cache-Control']
'max-age=187200'
```

1.3.5 Cookies

Cookies (and the Set-Cookie header) are handled with a couple methods. Most importantly:

```
>>> res.set_cookie('key', 'value', max_age=360, path='/',
...                domain='example.org', secure=True)
>>> res.headers['Set-Cookie']
'key=value; Domain=example.org; Max-Age=360; Path=/; expires=... GMT; secure'
>>> # To delete a cookie previously set in the client:
>>> res.delete_cookie('bad_cookie')
>>> res.headers['Set-Cookie']
'bad_cookie=; Max-Age=0; Path=/; expires=... GMT'
```

The only other real method of note (note that this does *not* delete the cookie from clients, only from the response object):

```
>>> res.unset_cookie('key')
>>> res.unset_cookie('bad_cookie')
>>> print res.headers.get('Set-Cookie')
None
```

1.3.6 Binding a Request

You can bind a request (or request WSGI environ) to the response object. This is available through `res.request` or `res.environ`. This is currently only used in setting `res.location`, to make the location absolute if necessary.

1.3.7 Response as a WSGI application

A response is a WSGI application, in that you can do:

```
>>> req = Request.blank('/')
>>> status, headers, app_iter = req.call_application(res)
```

A possible pattern for your application might be:

```
>>> def my_app(environ, start_response):
...     req = Request(environ)
...     res = Response()
...     res.content_type = 'text/plain'
...     parts = []
...     for name, value in sorted(req.environ.items()):
...         parts.append('%s: %r' % (name, value))
...     res.body = '\n'.join(parts)
...     return res(environ, start_response)
>>> req = Request.blank('/')
```

```
>>> res = req.get_response(my_app)
>>> print res
200 OK
Content-Type: text/plain; charset=UTF-8
Content-Length: ...

HTTP_HOST: 'localhost:80'
PATH_INFO: '/'
QUERY_STRING: ''
REQUEST_METHOD: 'GET'
SCRIPT_NAME: ''
SERVER_NAME: 'localhost'
SERVER_PORT: '80'
SERVER_PROTOCOL: 'HTTP/1.0'
wsgi.errors: <open file '<stderr>', mode 'w' at ...>
wsgi.input: <...IO... object at ...>
wsgi.multiprocess: False
wsgi.multithread: False
wsgi.run_once: False
wsgi.url_scheme: 'http'
wsgi.version: (1, 0)
```

1.4 Exceptions

In addition to Request and Response objects, there are a set of Python exceptions for different HTTP responses (3xx, 4xx, 5xx codes).

These provide a simple way to provide these non-200 response. A very simple body is provided.

```
>>> from webob.exc import *
>>> exc = HTTPTemporaryRedirect(location='foo')
>>> req = Request.blank('/path/to/something')
>>> print str(req.get_response(exc)).strip()
307 Temporary Redirect
Location: http://localhost/path/to/foo
Content-Length: 126
Content-Type: text/plain; charset=UTF-8

307 Temporary Redirect

The resource has been moved to http://localhost/path/to/foo; you should be redirected automatically.
```

Note that only if there's an `Accept: text/html` header in the request will an HTML response be given:

```
>>> req.accept += 'text/html'
>>> print str(req.get_response(exc)).strip()
307 Temporary Redirect
Location: http://localhost/path/to/foo
Content-Length: 270
Content-Type: text/html; charset=UTF-8

<html>
 <head>
  <title>307 Temporary Redirect</title>
 </head>
 <body>
  <h1>307 Temporary Redirect</h1>
```

```
  The resource has been moved to <a href="http://localhost/path/to/foo">http://localhost/path/to/foo</
you should be redirected automatically.


 </body>
</html>
```

This is taken from paste.httpexceptions, and if you have Paste installed then these exceptions will be subclasses of the Paste exceptions.

1.4.1 Conditional WSGI Application

The Response object can handle your conditional responses for you, checking If-None-Match, If-Modified-Since, and Range/If-Range.

To enable this you must create the response like `Response(conditional_response=True)`, or make a subclass like:

```
>>> class AppResponse(Response):
...     default_content_type = 'text/html'
...     default_conditional_response = True
>>> res = AppResponse(body='0123456789',
...                   last_modified=datetime(2005, 1, 1, 12, 0, tzinfo=UTC))
>>> req = Request.blank('/')
>>> req.if_modified_since = datetime(2006, 1, 1, 12, 0, tzinfo=UTC)
>>> req.get_response(res)
<Response ... 304 Not Modified>
>>> del req.if_modified_since
>>> res.etag = 'opaque-tag'
>>> req.if_none_match = 'opaque-tag'
>>> req.get_response(res)
<Response ... 304 Not Modified>

>>> req.if_none_match = '*'
>>> 'x' in req.if_none_match
True
>>> req.if_none_match = req.if_none_match
>>> 'x' in req.if_none_match
True
>>> req.if_none_match = None
>>> 'x' in req.if_none_match
False
>>> req.if_match = None
>>> 'x' in req.if_match
True
>>> req.if_match = req.if_match
>>> 'x' in req.if_match
True
>>> req.headers.get('If-Match')
'*'

>>> del req.if_none_match

>>> req.range = (1, 5)
>>> result = req.get_response(res)
>>> result.headers['content-range']
'bytes 1-4/10'
```

```
>>> result.body
'1234'
```

CHAPTER 2

Differences Between WebOb and Other Systems

This document points out some of the API differences between the Request and Response object, and the objects in other systems.

Contents

2.1 paste.wsgiwrappers and Pylons

The Pylons `request` and `response` object are based on `paste.wsgiwrappers.WSGIRequest` and `WSGIResponse`

There is no concept of `defaults` in WebOb. In Paste/Pylons these serve as threadlocal settings that control certain policies on the request and response object. In WebOb you should make your own subclasses to control policy (though in many ways simply being explicit elsewhere removes the need for this policy).

2.1.1 Request

`body`: This is a file-like object in WSGIRequest. In WebOb it is a string (to match Response.body) and the file-like object is available through `req.body_file`

`languages()`: This is available through `req.accept_language`, particularly `req.accept_language.best_match(supported_languages)`

`match_accept(mimetypes)`: This is available through `req.accept.first_match(mimetypes)`; or if you trust the client's quality ratings, you can use `req.accept.best_match(mimetypes)`

`errors`: This controls how unicode decode errors are handled; it is now named `unicode_errors`

There are also many extra methods and attributes on WebOb Request objects.

2.1.2 Response

`determine_charset()`: Is now available as `res.charset`

`has_header(header)`: Should be done with `header in res.headers`

`get_content()` and `wsgi_response()`: These are gone; you should use `res.body` or `res(environ, start_response)`

`write(content)`: Available in `res.body_file.write(content)`.

`flush()` and `tell()`: Not available.

There are also many extra methods and attributes on WebOb Response objects.

2.2 Django

This is a quick summary from reading the Django documentation.

2.2.1 Request

`encoding`: Is `req.charset`

`REQUEST`: Is `req.params`

`FILES`: File uploads are `cgi.FieldStorage` objects directly in `res.POST`

`META`: Is `req.environ`

`user`: No equivalent (too connected to application model for WebOb). There is `req.remote_user`, which is only ever a string.

`session`: No equivalent

raw_post_data: Available with `req.body`

__getitem__(key): You have to use `req.params`

is_secure(): No equivalent; you could use `req.scheme == 'https'`.

2.2.2 QueryDict

QueryDict is the way Django represents the multi-key dictionary-like objects that are request variables (query string and POST body variables). The equivalent in WebOb is MultiDict.

Mutability: WebOb dictionaries are sometimes mutable (req.GET is, req.params is not)

Ordering: I believe Django does not order the keys fully; MultiDict is a full ordering. Methods that iterate over the parameters iterate over keys in their order in the original request.

keys(), items(), values() (plus iter*): These return all values in MultiDict, but only the last value for a QueryDict. That is, given `a=1&a=2` with MultiDict `d.items()` returns `[('a', '1'), ('a', '2')]`, but QueryDict returns `[('a', '1')]`

getlist(key): Available as `d.getall(key)`

setlist(key): No direct equivalent

appendlist(key, value): Available as `d.add(key, value)`

setlistdefault(key, default_list): No direct equivalent

lists(): Is `d.dict_of_lists()`

The MultiDict object has a `d.getone(key)` method, that raises KeyError if there is not exactly one key. There is a method `d.mixed()` which returns a version where values are lists *if* there are multiple values for a list. This is similar to how many cgi-based request forms are represented.

2.2.3 Response

Constructor: Somewhat different. WebOb takes any keyword arguments as attribute assignments. Django only takes a couple arguments. The `mimetype` argument is `content_type`, and `content_type` is the entire `Content-Type` header (including charset).

dictionary-like: The Django response object is somewhat dictionary-like, setting headers. The equivalent dictionary-like object is `res.headers`. In WebOb this is a MultiDict.

has_header(header): Use `header in res.headers`

flush(),tell(): Not available

content: Use `res.body` for the `str` value, `res.text` for the `unicode` value

2.2.4 Response Subclasses

These are generally like `webob.exc` objects. `HttpResponseNotModified` is `HTTPNotModified`; this naming translation generally works.

2.3 CherryPy/TurboGears

The CherryPy request object is also used by TurboGears 1.x.

2.3.1 Request

app: No equivalent

base: `req.application_url`

close(): No equivalent

closed: No equivalent

config: No equivalent

cookie: A `SimpleCookie` object in CherryPy; a dictionary in WebOb (`SimpleCookie` can represent cookie parameters, but cookie parameters are only sent with responses not requests)

dispatch: No equivalent (this is the object dispatcher in CherryPy).

error_page, error_response, handle_error: No equivalent

get_resource(): Similar to `req.get_response(app)`

handler: No equivalent

headers, header_list: The WSGI environment represents headers as a dictionary, available through `req.headers` (no list form is available in the request).

hooks: No equivalent

local: No equivalent

methods_with_bodies: This represents methods where CherryPy will automatically try to read the request body. WebOb lazily reads POST requests with the correct content type, and no other bodies.

namespaces: No equivalent

protocol: As `req.environ['SERVER_PROTOCOL']`

query_string: As `req.query_string`

remote: `remote.ip` is like `req.remote_addr`. `remote.port` is not available. `remote.name` is in `req.environ.get('REMOTE_HOST')`

request_line: No equivalent

respond(): A method that is somewhat similar to `req.get_response()`.

rfile: `req.body_file`

run: No equivalent

server_protocol: As `req.environ['SERVER_PROTOCOL']`

show_tracebacks: No equivalent

throw_errors: No equivalent

throws: No equivalent

toolmaps: No equivalent

wsgi_environ: As `req.environ`

2.3.2 Response

From information from the wiki.

body: This is an iterable in CherryPy, a string in WebOb; `res.app_iter` gives an iterable in WebOb.

check_timeout: No equivalent

collapse_body(): This turns a stream/iterator body into a single string. Accessing `res.body` will do this automatically.

cookie: Accessible through `res.set_cookie(...)`, `res.delete_cookie`, `res.unset_cookie()`

finalize(): No equivalent

header_list: In `res.headerlist`

stream: This can make CherryPy stream the response body out directory. There is direct no equivalent; you can use a dynamically generated iterator to do something similar.

time: No equivalent

timed_out: No equivalent

2.4 Yaro

Yaro is a small wrapper around the WSGI environment, much like WebOb in scope.

The WebOb objects have many more methods and attributes. The Yaro Response object is a much smaller subset of WebOb's Response.

2.4.1 Request

query: As `req.GET`

form: As `req.POST`

cookie: A `SimpleCookie` object in Yaro; a dictionary in WebOb (`SimpleCookie` can represent cookie parameters, but cookie parameters are only sent with responses not requests)

uri: Returns a URI object, no equivalent (only string URIs available).

redirect: Not available (response-related). `webob.exc.HTTPFound()` can be useful here.

forward(yaroapp),wsgi_forward(wsgiapp): Available with `req.get_response(app)` and `req.call_application(app)`. In both cases it is a WSGI application in WebOb, there is no special kind of communication; `req.call_application()` just returns a `webob.Response` object.

res: The request object in WebOb *may* have a `req.response` attribute.

2.5 Werkzeug

An offshoot of Pocoo, this library is based around WSGI, similar to Paste and Yaro.

This is taken from the wrapper documentation.

2.5.1 Request

path: As `req.path_info`

args: As `req.GET`

form: As `req.POST`

values: As `req.params`

files: In `req.POST` (as FieldStorage objects)

data: In `req.body_file`

2.5.2 Response

response: In `res.body` (settable as `res.body` or `res.app_iter`)

status: In `res.status_code`

mimetype: In `res.content_type`

2.6 Zope 3

From the Zope 3 interfaces for the Request and Response.

2.6.1 Request

`locale`, `setupLocale()`: This is not fully calculated, but information is available in `req.accept_languages`.

`principal`, `setPrincipal(principal)`: `req.remote_user` gives the username, but there is no standard place for a user *object*.

`publication`, `setPublication()`, These are associated with the object publishing system in Zope. This kind of publishing system is outside the scope of WebOb.

`traverse(object)`, `getTraversalStack()`, `setTraversalStack()`: These all relate to traversal, which is part of the publishing system.

`processInputs()`, `setPathSuffix(steps)`: Also associated with traversal and preparing the request.

`environment`: In `req.environ`

`bodyStream`: In `req.body_file`

`interaction`: This is the security context for the request; all the possible participants or principals in the request. There's no equivalent.

`annotations`: Extra information associated with the request. This would generally go in custom keys of `req.environ`, or if you set attributes those attributes are stored in `req.environ['webob.adhoc_attrs']`.

`debug`: There is no standard debug flag for WebOb.

`__getitem__(key)`, `get(key)`, etc: These treat the request like a dictionary, which WebOb does not do. They seem to take values from the environment, not parameters. Also on the Zope request object is `items()`, `__contains__(key)`, `__iter__()`, `keys()`, `__len__()`, `values()`.

`getPositionalArguments()`: I'm not sure what the equivalent would be, as there are no positional arguments during instantiation (it doesn't fit into WSGI). Maybe `wsgiorg.urlvars`?

`retry()`, `supportsRetry()`: Creates a new request that can be used to retry a request. Similar to `req.copy()`.

`close()`, `hold(obj)`: This closes resources associated with the request, including any "held" objects. There's nothing similar.

2.6.2 Response

authUser: Not sure what this is or does.

reset(): No direct equivalent; you'd have to do `res.headers = []; res.body = ''; res.status = 200`

setCookie(name, value, **kw): Is `res.set_cookie(...)`.

getCookie(name): No equivalent. Hm.

expireCookie(name): Is `res.delete_cookie(name)`.

appendToCookie(name, value): This appends the value to any existing cookie (separating values with a colon). WebOb does not do this.

setStatus(status): Availble by setting `res.status` (can be set to an integer or a string of "code reason").

getHeader(name, default=None): Is `res.headers.get(name)`.

getStatus(): Is `res.status_code` (or `res.status` to include reason)

addHeader(name, value): Is `res.headers.add(name, value)` (in Zope and WebOb, this does not clobber any previous value).

getHeaders(): Is `res.headerlist`.

setHeader(name, value): Is `res.headers[name] = value`.

getStatusString(): Is `res.status`.

consumeBody(): This consumes any non-string body to turn the body into a single string. Any access to `res.body` will do this (e.g., when you have set the `res.app_iter`).

internalError(): This is available with `webob.exc.HTTP*()`.

handleException(exc_info): This is provided with a tool like `paste.exceptions`.

consumeBodyIter(): This returns the iterable for the body, even if the body was a string. Anytime you access `res.app_iter` you will get an iterable. `res.body` and `res.app_iter` can be interchanged and accessed as many times as you want, unlike the Zope equivalents.

setResult(result): You can achieve the same thing through `res.body = result`, or `res.app_iter = result`. `res.body` accepts None, a unicode string (*if* you have set a charset) or a normal string. `res.app_iter` only accepts None and an interable. You can't update all of a response with one call.

Like in Zope, WebOb updates Content-Length. Unlike Zope, it does not automatically calculate a charset.

2.7 mod_python

Some key attributes from the mod_python request object.

2.7.1 Request

req.uri: In `req.path`.

req.user: In `req.remote_user`.

req.get_remote_host(): In `req.environ['REMOTE_ADDR']` or `req.remote_addr`.

req.headers_in.get('referer'): In `req.headers.get('referer')` or `req.referer` (same pattern for other request headers, presumably).

2.7.2 Response

`util.redirect` or `req.status = apache.HTTP_MOVED_TEMPORARILY`:

```
from webob.exc import HTTPTemporaryRedirect
exc = HTTPTemporaryRedirect(location=url)
return exc(environ, start_response)
```

`req.content_type = "application/x-csv"` and `req.headers_out.add('Content-Disposition', 'attachment;filename=somefile.csv')`:

```
res = req.ResponseClass()
res.content_type = 'application/x-csv'
res.headers.add('Content-Disposition', 'attachment;filename=somefile.csv')
return res(environ, start_response)
```

2.8 webapp Response

The Google App Engine webapp framework uses the WebOb Request object, but does not use its Response object.

The constructor for `webapp.Response` does not take any arguments. The response is created by the framework, so you don't use it like `return Response(...)`, instead you use `self.response`. Also the response object automatically has `Cache-Control: no-cache` set, while the WebOb response does not set any cache headers.

`resp.set_status(code, message=None)`: This is handled by setting the `resp.status` attribute.

`resp.clear()`: You'd do `resp.body = ""`

`resp.wsgi_write(start_response)`: This writes the response using the `start_response` callback, and using the `start_response` writer. The WebOb response object is called as a WSGI app (`resp(environ, start_response)`) to do the equivalent.

`resp.out.write(text)`: This writes to an internal `StringIO` instance of the response. This uses the ability of the standard StringIO object to hold either unicode or `str` text, and so long as you are always consistent it will encode your content (but it does not respect your preferred encoding, it always uses UTF-8). The WebOb method `resp.write(text)` is basically equivalent, and also accepts unicode (using `resp.charset` for the encoding). You can also write to `resp.body_file`, but it does not allow unicode.

Besides exposing a `.headers` attribute (based on wsgiref.headers.Headers) there is no other API for the webapp response object. This means the response lacks:

- A usefully readable body or status.
- A useful constructor that makes it easy to treat responses like objects.
- Providing a non-string `app_iter` for the body (like a generator).
- Parsing of the Content-Type charset.
- Getter/setters for parsed forms of headers, specifically cache_control and last_modified.
- The `cache_expires` method
- `set_cookie`, `delete_cookie`, and `unset_cookie`. Instead you have to simply manually set the Set-Cookie header.
- `encode_content` and `decode_content` for handling gzip encoding.
- `md5_etag()` for generating an etag from the body.
- Conditional responses that will return 304 based on the response and request headers.

- The ability to serve Range request automatically.

2.9 PHP

PHP does not have anything really resembling a request and response object. Instead these are encoded in a set of global objects for the request and functions for the response.

2.9.1 $_POST, $_GET, $_FILES

These represent `req.POST` and `req.GET`.

PHP uses the variable names to tell whether a variable can hold multiple values. For instance `$_POST['name[]']`, which will be an array. In WebOb any variable can have multiple values, and you can get these through `req.POST.getall('name')`.

The files in `$_FILES` are simply in `req.POST` in WebOb, as FieldStorage instances.

2.9.2 $_COOKIES

This is in `req.cookies`.

2.9.3 $_SERVER, $_REQUEST, $_ENV

These are all in `req.environ`. These are not split up like they are in PHP, it's all just one dictionary. Everything that would typically be in `$_ENV` is technically optional, and outside of a couple CGI-standard keys in `$_SERVER` most of those are also optional, but it is common for WSGI servers to populate the request with similar information as PHP.

2.9.4 $HTTP_RAW_POST_DATA

This contains the unparsed data in the request body. This is in `req.body`.

2.9.5 The response

Response headers in PHP are sent with `header("Header-Name: value")`. In WebOb there is a dictionary in `resp.headers` that can have values set; the headers aren't actually sent until you send the response. You can add headers without overwriting (the equivalent of `header("...", false)`) with `resp.headers.add('Header-Name', 'value')`.

The status in PHP is sent with `http_send_status(code)`. In WebOb this is `resp.status = code`.

The body in PHP is sent implicitly through the rendering of the PHP body (or with `echo` or any other functions that send output).

CHAPTER 3

License

CHAPTER 4

API Documentation

Reference material for every public API exposed by WebOb:

4.1 webob.client – Send WSGI requests over HTTP

4.1.1 Client

class webob.client.**SendRequest**(*HTTPConnection=<class httplib.HTTPConnection>*, *HTTPSConnection=<class httplib.HTTPSConnection>*)

Sends the request, as described by the environ, over actual HTTP. All controls about how it is sent are contained in the request environ itself.

This connects to the server given in SERVER_NAME:SERVER_PORT, and sends the Host header in HTTP_HOST – they do not have to match. You can send requests to servers despite what DNS says.

Set `environ['webob.client.timeout'] = 10` to set the timeout on the request (to, for example, 10 seconds).

Does not add X-Forwarded-For or other standard headers

If you use `send_request_app` then simple `httplib` connections will be used.

parse_headers(*message*)

Turn a Message object into a list of WSGI-style headers.

webob.client.**send_request_app**

4.2 webob.cookies – Cookies

4.2.1 Cookies

class webob.cookies.**CookieProfile**(*cookie_name*, *secure=False*, *max_age=None*, *httponly=None*, *path='/'*, *domains=None*, *serializer=None*)

A helper class that helps bring some sanity to the insanity that is cookie handling.

The helper is capable of generating multiple cookies if necessary to support subdomains and parent domains.

cookie_name The name of the cookie used for sessioning. Default: `'session'`.

max_age The maximum age of the cookie used for sessioning (in seconds). Default: `None` (browser scope).

secure The 'secure' flag of the session cookie. Default: `False`.

httponly Hide the cookie from Javascript by setting the 'HttpOnly' flag of the session cookie. Default: `False`.

path The path used for the session cookie. Default: `'/'`.

domains The domain(s) used for the session cookie. Default: `None` (no domain). Can be passed an iterable containing multiple domains, this will set multiple cookies one for each domain.

serializer An object with two methods: `loads` and `dumps`. The `loads` method should accept a bytestring and return a Python object. The `dumps` method should accept a Python object and return bytes. A `ValueError` should be raised for malformed inputs. Default: `None`, which will use a derivation of `json.dumps()` and `json.loads()`.

bind(*request*)
Bind a request to a copy of this instance and return it

get_headers(*value*, *domains=<object object>*, *max_age=<object object>*, *path=<object object>*, *secure=<object object>*, *httponly=<object object>*)
Retrieve raw headers for setting cookies.

Returns a list of headers that should be set for the cookies to be correctly tracked.

get_value()
Looks for a cookie by name in the currently bound request, and returns its value. If the cookie profile is not bound to a request, this method will raise a `ValueError`.

Looks for the cookie in the cookies jar, and if it can find it it will attempt to deserialize it. Returns `None` if there is no cookie or if the value in the cookie cannot be successfully deserialized.

set_cookies(*response*, *value*, *domains=<object object>*, *max_age=<object object>*, *path=<object object>*, *secure=<object object>*, *httponly=<object object>*)
Set the cookies on a response.

class webob.cookies.**SignedCookieProfile**(*secret*, *salt*, *cookie_name*, *secure=False*, *max_age=None*, *httponly=False*, *path='/'*, *domains=None*, *hashalg='sha512'*, *serializer=None*)
A helper for generating cookies that are signed to prevent tampering.

By default this will create a single cookie, given a value it will serialize it, then use HMAC to cryptographically sign the data. Finally the result is base64-encoded for transport. This way a remote user can not tamper with the value without uncovering the secret/salt used.

secret A string which is used to sign the cookie. The secret should be at least as long as the block size of the selected hash algorithm. For `sha512` this would mean a 128 bit (64 character) secret.

salt A namespace to avoid collisions between different uses of a shared secret.

hashalg The HMAC digest algorithm to use for signing. The algorithm must be supported by the `hashlib` library. Default: `'sha512'`.

cookie_name The name of the cookie used for sessioning. Default: `'session'`.

max_age The maximum age of the cookie used for sessioning (in seconds). Default: `None` (browser scope).

secure The 'secure' flag of the session cookie. Default: `False`.

httponly Hide the cookie from Javascript by setting the 'HttpOnly' flag of the session cookie. Default: `False`.

path The path used for the session cookie. Default: `'/'`.

domains The domain(s) used for the session cookie. Default: `None` (no domain). Can be passed an iterable containing multiple domains, this will set multiple cookies one for each domain.

serializer An object with two methods: *loads‘* and `dumps`. The `loads` method should accept bytes and return a Python object. The `dumps` method should accept a Python object and return bytes. A `ValueError` should be raised for malformed inputs. Default: `None‘, which will use a derivation of :func:‘json.dumps‘ and ‘‘json.loads`.

bind(*request*)
Bind a request to a copy of this instance and return it

class `webob.cookies.`**SignedSerializer**(*secret*, *salt*, *hashalg='sha512'*, *serializer=None*)
A helper to cryptographically sign arbitrary content using HMAC.

The serializer accepts arbitrary functions for performing the actual serialization and deserialization.

secret A string which is used to sign the cookie. The secret should be at least as long as the block size of the selected hash algorithm. For `sha512` this would mean a 128 bit (64 character) secret.

salt A namespace to avoid collisions between different uses of a shared secret.

hashalg The HMAC digest algorithm to use for signing. The algorithm must be supported by the `hashlib` library. Default: `'sha512'`.

serializer An object with two methods: *loads‘* and `dumps`. The `loads` method should accept bytes and return a Python object. The `dumps` method should accept a Python object and return bytes. A `ValueError` should be raised for malformed inputs. Default: `None‘, which will use a derivation of :func:‘json.dumps‘ and ‘‘json.loads`.

dumps(*appstruct*)
Given an `appstruct`, serialize and sign the data.

Returns a bytestring.

loads(*bstruct*)
Given a `bstruct` (a bytestring), verify the signature and then deserialize and return the deserialized value.

A `ValueError` will be raised if the signature fails to validate.

class `webob.cookies.`**JSONSerializer**
A serializer which uses *json.dumps‘* and `json.loads`

`webob.cookies.`**make_cookie**(*name*, *value*, *max_age=None*, *path='/'*, *domain=None*, *secure=False*, *httponly=False*, *comment=None*)
Generate a cookie value. If `value` is None, generate a cookie value with an expiration date in the past

4.3 webob.dec – WSGIfy decorator

Decorators to wrap functions to make them WSGI applications.

The main decorator *wsgify* turns a function into a WSGI application (while also allowing normal calling of the method with an instantiated request).

4.3.1 Decorator

class `webob.dec.`**wsgify**(*func=None*, *RequestClass=None*, *args=()*, *kwargs=None*, *middleware_wraps=None*)
Turns a request-taking, response-returning function into a WSGI app

You can use this like:

```
@wsgify
def myfunc(req):
    return webob.Response('hey there')
```

With that `myfunc` will be a WSGI application, callable like `app_iter = myfunc(environ, start_response)`. You can also call it like normal, e.g., `resp = myfunc(req)`. (You can also wrap methods, like `def myfunc(self, req)`.)

If you raise exceptions from *webob.exc* they will be turned into WSGI responses.

There are also several parameters you can use to customize the decorator. Most notably, you can use a `webob.Request` subclass, like:

```
class MyRequest(webob.Request):
    @property
    def is_local(self):
        return self.remote_addr == '127.0.0.1'
@wsgify(RequestClass=MyRequest)
def myfunc(req):
    if req.is_local:
        return Response('hi!')
    else:
        raise webob.exc.HTTPForbidden
```

Another customization you can add is to add *args* (positional arguments) or *kwargs* (of course, keyword arguments). While generally not that useful, you can use this to create multiple WSGI apps from one function, like:

```
import simplejson
def serve_json(req, json_obj):
    return Response(json.dumps(json_obj),
                    content_type='application/json')

serve_ob1 = wsgify(serve_json, args=(ob1,))
serve_ob2 = wsgify(serve_json, args=(ob2,))
```

You can return several things from a function:

- A `webob.Response` object (or subclass)
- *Any* WSGI application
- None, and then `req.response` will be used (a pre-instantiated Response object)
- A string, which will be written to `req.response` and then that response will be used.
- Raise an exception from *webob.exc*

Also see *wsgify.middleware()* for a way to make middleware.

You can also subclass this decorator; the most useful things to do in a subclass would be to change *RequestClass* or override *call_func* (e.g., to add `req.urlvars` as keyword arguments to the function).

RequestClass
alias of `Request`

call_func(*req*, **args*, ***kwargs*)
Call the wrapped function; override this in a subclass to change how the function is called.

clone(*func=None*, ***kw*)
Creates a copy/clone of this object, but with some parameters rebound

get (*url*, ***kw*)
Run a GET request on this application, returning a Response.

This creates a request object using the given URL, and any other keyword arguments are set on the request object (e.g., `last_modified=datetime.now()`).

```
resp = myapp.get('/article?id=10')
```

classmethod middleware (*middle_func=None*, *app=None*, ***kw*)
Creates middleware

Use this like:

```
@wsgify.middleware
def restrict_ip(req, app, ips):
    if req.remote_addr not in ips:
        raise webob.exc.HTTPForbidden('Bad IP: %s' % req.remote_addr)
    return app

@wsgify
def app(req):
    return 'hi'

wrapped = restrict_ip(app, ips=['127.0.0.1'])
```

Or if you want to write output-rewriting middleware:

```
@wsgify.middleware
def all_caps(req, app):
    resp = req.get_response(app)
    resp.body = resp.body.upper()
    return resp

wrapped = all_caps(app)
```

Note that you must call `req.get_response(app)` to get a WebOb response object. If you are not modifying the output, you can just return the app.

As you can see, this method doesn't actually create an application, but creates "middleware" that can be bound to an application, along with "configuration" (that is, any other keyword arguments you pass when binding the application).

post (*url*, *POST=None*, ***kw*)
Run a POST request on this application, returning a Response.

The second argument (*POST*) can be the request body (a string), or a dictionary or list of two-tuples, that give the POST body.

```
resp = myapp.post('/article/new',
                  dict(title='My Day',
                       content='I ate a sandwich'))
```

request (*url*, ***kw*)
Run a request on this application, returning a Response.

This can be used for DELETE, PUT, etc requests. E.g.:

```
resp = myapp.request('/article/1', method='PUT', body='New article')
```

4.4 webob.exc – WebOb Exceptions

This module processes Python exceptions that relate to HTTP exceptions by defining a set of exceptions, all subclasses of HTTPException. Each exception, in addition to being a Python exception that can be raised and caught, is also a WSGI application and `webob.Response` object.

This module defines exceptions according to RFC 2068 [1] : codes with 100-300 are not really errors; 400's are client errors, and 500's are server errors. According to the WSGI specification [2] , the application can call `start_response` more then once only under two conditions: (a) the response has not yet been sent, or (b) if the second and subsequent invocations of `start_response` have a valid `exc_info` argument obtained from `sys.exc_info()`. The WSGI specification then requires the server or gateway to handle the case where content has been sent and then an exception was encountered.

Exception

HTTPException

HTTPOk

- 200 - *HTTPOk*
- 201 - *HTTPCreated*
- 202 - *HTTPAccepted*
- 203 - *HTTPNonAuthoritativeInformation*
- 204 - *HTTPNoContent*
- 205 - *HTTPResetContent*
- 206 - *HTTPPartialContent*

HTTPRedirection

- 300 - *HTTPMultipleChoices*
- 301 - *HTTPMovedPermanently*
- 302 - *HTTPFound*
- 303 - *HTTPSeeOther*
- 304 - *HTTPNotModified*
- 305 - *HTTPUseProxy*
- 307 - *HTTPTemporaryRedirect*
- 308 - HTTPPermanentRedirect

HTTPError

HTTPClientError

- 400 - *HTTPBadRequest*
- 401 - *HTTPUnauthorized*
- 402 - *HTTPPaymentRequired*
- 403 - *HTTPForbidden*
- 404 - *HTTPNotFound*

[1] http://www.python.org/peps/pep-0333.html#error-handling
[2] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.5

- 405 - *HTTPMethodNotAllowed*
- 406 - *HTTPNotAcceptable*
- 407 - *HTTPProxyAuthenticationRequired*
- 408 - *HTTPRequestTimeout*
- 409 - *HTTPConflict*
- 410 - *HTTPGone*
- 411 - *HTTPLengthRequired*
- 412 - *HTTPPreconditionFailed*
- 413 - *HTTPRequestEntityTooLarge*
- 414 - *HTTPRequestURITooLong*
- 415 - *HTTPUnsupportedMediaType*
- 416 - *HTTPRequestRangeNotSatisfiable*
- 417 - *HTTPExpectationFailed*
- 422 - *HTTPUnprocessableEntity*
- 423 - *HTTPLocked*
- 424 - *HTTPFailedDependency*
- 428 - *HTTPPreconditionRequired*
- 429 - *HTTPTooManyRequests*
- 431 - *HTTPRequestHeaderFieldsTooLarge*
- 451 - *HTTPUnavailableForLegalReasons*

HTTPServerError

- 500 - *HTTPInternalServerError*
- 501 - *HTTPNotImplemented*
- 502 - *HTTPBadGateway*
- 503 - *HTTPServiceUnavailable*
- 504 - *HTTPGatewayTimeout*
- 505 - *HTTPVersionNotSupported*
- 511 - *HTTPNetworkAuthenticationRequired*

4.4.1 Usage notes

The HTTPException class is complicated by 4 factors:

1. The content given to the exception may either be plain-text or as html-text.
2. The template may want to have string-substitutions taken from the current `environ` or values from incoming headers. This is especially troublesome due to case sensitivity.
3. The final output may either be text/plain or text/html mime-type as requested by the client application.
4. Each exception has a default explanation, but those who raise exceptions may want to provide additional detail.

Subclass attributes and call parameters are designed to provide an easier path through the complications.

Attributes:

code the HTTP status code for the exception

title remainder of the status line (stuff after the code)

explanation a plain-text explanation of the error message that is not subject to environment or header substitutions; it is accessible in the template via %(explanation)s

detail a plain-text message customization that is not subject to environment or header substitutions; accessible in the template via %(detail)s

body_template a content fragment (in HTML) used for environment and header substitution; the default template includes both the explanation and further detail provided in the message

Parameters:

detail a plain-text override of the default `detail`

headers a list of (k,v) header pairs

comment a plain-text additional information which is usually stripped/hidden for end-users

body_template a string.Template object containing a content fragment in HTML that frames the explanation and further detail

To override the template (which is HTML content) or the plain-text explanation, one must subclass the given exception; or customize it after it has been created. This particular breakdown of a message into explanation, detail and template allows both the creation of plain-text and html messages for various clients as well as error-free substitution of environment variables and headers.

The subclasses of *_HTTPMove* (*HTTPMultipleChoices*, *HTTPMovedPermanently*, *HTTPFound*, *HTTPSeeOther*, *HTTPUseProxy* and *HTTPTemporaryRedirect*) are redirections that require a `Location` field. Reflecting this, these subclasses have two additional keyword arguments: `location` and `add_slash`.

Parameters:

location to set the location immediately

add_slash set to True to redirect to the same URL as the request, except with a `/` appended

Relative URLs in the location will be resolved to absolute.

References:

4.4.2 HTTP Exceptions

exception `webob.exc.`**`HTTPException`**(*message*, *wsgi_response*)

exception `webob.exc.`**`WSGIHTTPException`**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

exception `webob.exc.`**`HTTPError`**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

base class for status codes in the 400's and 500's

This is an exception which indicates that an error has occurred, and that any work in progress should not be committed. These are typically results in the 400's and 500's.

exception `webob.exc.`**`HTTPRedirection`**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

base class for 300's status code (redirections)

This is an abstract base class for 3xx redirection. It indicates that further action needs to be taken by the user agent in order to fulfill the request. It does not necessarly signal an error condition.

exception webob.exc.**HTTPOk**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

Base class for the 200's status code (successful responses)

code: 200, title: OK

exception webob.exc.**HTTPCreated**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPOk*

This indicates that request has been fulfilled and resulted in a new resource being created.

code: 201, title: Created

exception webob.exc.**HTTPAccepted**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPOk*

This indicates that the request has been accepted for processing, but the processing has not been completed.

code: 202, title: Accepted

exception webob.exc.**HTTPNonAuthoritativeInformation**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPOk*

This indicates that the returned metainformation in the entity-header is not the definitive set as available from the origin server, but is gathered from a local or a third-party copy.

code: 203, title: Non-Authoritative Information

exception webob.exc.**HTTPNoContent**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPOk*

This indicates that the server has fulfilled the request but does not need to return an entity-body, and might want to return updated metainformation.

code: 204, title: No Content

exception webob.exc.**HTTPResetContent**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPOk*

This indicates that the the server has fulfilled the request and the user agent SHOULD reset the document view which caused the request to be sent.

code: 205, title: Reset Content

exception webob.exc.**HTTPPartialContent**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPOk*

This indicates that the server has fulfilled the partial GET request for the resource.

code: 206, title: Partial Content

exception webob.exc.**_HTTPMove**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, *location=None*, *add_slash=False*)

redirections which require a Location field

Since a 'Location' header is a required attribute of 301, 302, 303, 305, 307 and 308 (but not 304), this base class provides the mechanics to make this easy.

You can provide a location keyword argument to set the location immediately. You may also give `add_slash=True` if you want to redirect to the same URL as the request, except with a / added to the end.

Relative URLs in the location will be resolved to absolute.

exception webob.exc.**HTTPMultipleChoices**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, *location=None*, *add_slash=False*)

subclass of *_HTTPMove*

This indicates that the requested resource corresponds to any one of a set of representations, each with its own specific location, and agent-driven negotiation information is being provided so that the user can select a preferred representation and redirect its request to that location.

code: 300, title: Multiple Choices

exception webob.exc.**HTTPMovedPermanently**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, *location=None*, *add_slash=False*)

subclass of *_HTTPMove*

This indicates that the requested resource has been assigned a new permanent URI and any future references to this resource SHOULD use one of the returned URIs.

code: 301, title: Moved Permanently

exception webob.exc.**HTTPFound**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, *location=None*, *add_slash=False*)

subclass of *_HTTPMove*

This indicates that the requested resource resides temporarily under a different URI.

code: 302, title: Found

exception webob.exc.**HTTPSeeOther**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, *location=None*, *add_slash=False*)

subclass of *_HTTPMove*

This indicates that the response to the request can be found under a different URI and SHOULD be retrieved using a GET method on that resource.

code: 303, title: See Other

exception webob.exc.**HTTPNotModified**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPRedirection*

This indicates that if the client has performed a conditional GET request and access is allowed, but the document has not been modified, the server SHOULD respond with this status code.

code: 304, title: Not Modified

exception webob.exc.**HTTPUseProxy**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, *location=None*, *add_slash=False*)

subclass of *_HTTPMove*

This indicates that the requested resource MUST be accessed through the proxy given by the Location field.

code: 305, title: Use Proxy

exception webob.exc.**HTTPTemporaryRedirect**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, *location=None*, *add_slash=False*)

subclass of *_HTTPMove*

This indicates that the requested resource resides temporarily under a different URI.

code: 307, title: Temporary Redirect

exception webob.exc.**HTTPClientError**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

base class for the 400's, where the client is in error

This is an error condition in which the client is presumed to be in-error. This is an expected problem, and thus is not considered a bug. A server-side traceback is not warranted. Unless specialized, this is a '400 Bad Request'

code: 400, title: Bad Request

exception webob.exc.**HTTPBadRequest**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

exception webob.exc.**HTTPUnauthorized**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPClientError*

This indicates that the request requires user authentication.

code: 401, title: Unauthorized

exception webob.exc.**HTTPPaymentRequired**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPClientError*

code: 402, title: Payment Required

exception webob.exc.**HTTPForbidden**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPClientError*

This indicates that the server understood the request, but is refusing to fulfill it.

code: 403, title: Forbidden

exception webob.exc.**HTTPNotFound**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPClientError*

This indicates that the server did not find anything matching the Request-URI.

code: 404, title: Not Found

exception webob.exc.**HTTPMethodNotAllowed**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPClientError*

This indicates that the method specified in the Request-Line is not allowed for the resource identified by the Request-URI.

code: 405, title: Method Not Allowed

exception webob.exc.**HTTPNotAcceptable**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPClientError*

This indicates the resource identified by the request is only capable of generating response entities which have content characteristics not acceptable according to the accept headers sent in the request.

code: 406, title: Not Acceptable

exception webob.exc.**HTTPProxyAuthenticationRequired**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPClientError*

This is similar to 401, but indicates that the client must first authenticate itself with the proxy.

code: 407, title: Proxy Authentication Required

exception webob.exc.**HTTPRequestTimeout**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPClientError*

This indicates that the client did not produce a request within the time that the server was prepared to wait.

code: 408, title: Request Timeout

exception webob.exc.**HTTPConflict**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPClientError*

This indicates that the request could not be completed due to a conflict with the current state of the resource.

code: 409, title: Conflict

exception webob.exc.**HTTPGone**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPClientError*

This indicates that the requested resource is no longer available at the server and no forwarding address is known.

code: 410, title: Gone

exception webob.exc.**HTTPLengthRequired**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPClientError*

This indicates that the the server refuses to accept the request without a defined Content-Length.

code: 411, title: Length Required

exception webob.exc.**HTTPPreconditionFailed**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPClientError*

This indicates that the precondition given in one or more of the request-header fields evaluated to false when it was tested on the server.

code: 412, title: Precondition Failed

exception webob.exc.**HTTPRequestEntityTooLarge**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPClientError*

This indicates that the server is refusing to process a request because the request entity is larger than the server is willing or able to process.

code: 413, title: Request Entity Too Large

exception webob.exc.**HTTPRequestURITooLong**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPClientError*

This indicates that the server is refusing to service the request because the Request-URI is longer than the server is willing to interpret.

code: 414, title: Request-URI Too Long

exception webob.exc.**HTTPUnsupportedMediaType**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPClientError*

This indicates that the server is refusing to service the request because the entity of the request is in a format not supported by the requested resource for the requested method.

code: 415, title: Unsupported Media Type

exception webob.exc.**HTTPRequestRangeNotSatisfiable**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPClientError*

The server SHOULD return a response with this status code if a request included a Range request-header field, and none of the range-specifier values in this field overlap the current extent of the selected resource, and the request did not include an If-Range request-header field.

code: 416, title: Request Range Not Satisfiable

exception webob.exc.**HTTPExpectationFailed**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPClientError*

This indidcates that the expectation given in an Expect request-header field could not be met by this server.

code: 417, title: Expectation Failed

exception webob.exc.**HTTPUnprocessableEntity**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPClientError*

This indicates that the server is unable to process the contained instructions.

code: 422, title: Unprocessable Entity

exception webob.exc.**HTTPLocked**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPClientError*

This indicates that the resource is locked.

code: 423, title: Locked

exception webob.exc.**HTTPFailedDependency**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPClientError*

This indicates that the method could not be performed because the requested action depended on another action and that action failed.

code: 424, title: Failed Dependency

exception webob.exc.**HTTPPreconditionRequired**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPClientError*

This indicates that the origin server requires the request to be conditional. From RFC 6585, "Additional HTTP Status Codes".

code: 428, title: Precondition Required

exception webob.exc.**HTTPTooManyRequests**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPClientError*

This indicates that the client has sent too many requests in a given amount of time. Useful for rate limiting.

From RFC 6585, "Additional HTTP Status Codes".

code: 429, title: Too Many Requests

exception webob.exc.**HTTPRequestHeaderFieldsTooLarge**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPClientError*

This indicates that the server is unwilling to process the request because its header fields are too large. The request may be resubmitted after reducing the size of the request header fields.

From RFC 6585, "Additional HTTP Status Codes".

code: 431, title: Request Header Fields Too Large

exception webob.exc.**HTTPUnavailableForLegalReasons**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPClientError*

This indicates that the server is unable to process the request because of legal reasons, e.g. censorship or government-mandated blocked access.

From the draft "A New HTTP Status Code for Legally-restricted Resources" by Tim Bray:

http://tools.ietf.org/html/draft-tbray-http-legally-restricted-status-00

code: 451, title: Unavailable For Legal Reasons

exception webob.exc.**HTTPServerError**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

base class for the 500's, where the server is in-error

This is an error condition in which the server is presumed to be in-error. This is usually unexpected, and thus requires a traceback; ideally, opening a support ticket for the customer. Unless specialized, this is a '500 Internal Server Error'

exception webob.exc.**HTTPInternalServerError**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

exception webob.exc.**HTTPNotImplemented**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPServerError*

This indicates that the server does not support the functionality required to fulfill the request.

code: 501, title: Not Implemented

exception webob.exc.**HTTPBadGateway**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPServerError*

This indicates that the server, while acting as a gateway or proxy, received an invalid response from the upstream server it accessed in attempting to fulfill the request.

code: 502, title: Bad Gateway

exception webob.exc.**HTTPServiceUnavailable**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPServerError*

This indicates that the server is currently unable to handle the request due to a temporary overloading or maintenance of the server.

code: 503, title: Service Unavailable

exception webob.exc.**HTTPGatewayTimeout**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPServerError*

This indicates that the server, while acting as a gateway or proxy, did not receive a timely response from the upstream server specified by the URI (e.g. HTTP, FTP, LDAP) or some other auxiliary server (e.g. DNS) it needed to access in attempting to complete the request.

code: 504, title: Gateway Timeout

exception webob.exc.**HTTPVersionNotSupported**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPServerError*

This indicates that the server does not support, or refuses to support, the HTTP protocol version that was used in the request message.

code: 505, title: HTTP Version Not Supported

exception webob.exc.**HTTPInsufficientStorage**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPServerError*

This indicates that the server does not have enough space to save the resource.

code: 507, title: Insufficient Storage

exception webob.exc.**HTTPNetworkAuthenticationRequired**(*detail=None*, *headers=None*, *comment=None*, *body_template=None*, ***kw*)

subclass of *HTTPServerError*

This indicates that the client needs to authenticate to gain network access. From RFC 6585, "Additional HTTP Status Codes".

code: 511, title: Network Authentication Required

exception webob.exc.**HTTPExceptionMiddleware**(*application*)

Middleware that catches exceptions in the sub-application. This does not catch exceptions in the app_iter; only during the initial calling of the application.

This should be put *very close* to applications that might raise these exceptions. This should not be applied globally; letting *expected* exceptions raise through the WSGI stack is dangerous.

4.5 webob.multidict – multi-value dictionary object

Gives a multi-value dictionary object (MultiDict) plus several wrappers

class webob.multidict.**MultiDict**(**args*, ***kw*)

An ordered dictionary that can have multiple values for each key. Adds the methods getall, getone, mixed and extend and add to the normal dictionary interface.

add(*key*, *value*)

Add the key and value, not overwriting any previous value.

dict_of_lists()

Returns a dictionary where each key is associated with a list of values.

classmethod from_fieldstorage(*fs*)

Create a dict from a cgi.FieldStorage instance

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

getall(*key*)
Return a list of all values matching the key (may be an empty list)

getone(*key*)
Get one value matching the key, raising a KeyError if multiple values were found.

mixed()
Returns a dictionary where the values are either single values, or a list of values when a key/value appears more than once in this dictionary. This is similar to the kind of dictionary often used to represent the variables in a web request.

classmethod view_list(*lst*)
Create a dict that is a view on the given list

class webob.multidict.**NestedMultiDict**(**dicts*)
Wraps several MultiDict objects, treating it as one large MultiDict

class webob.multidict.**NoVars**(*reason=None*)
Represents no variables; used when no variables are applicable.

This is read-only

4.6 webob.request – Request

4.6.1 Request

class webob.request.**Request**(*environ*, *charset=None*, *unicode_errors=None*, *decode_param_names=None*, ***kw*)
The default request implementation

class webob.request.**BaseRequest**(*environ*, *charset=None*, *unicode_errors=None*, *decode_param_names=None*, ***kw*)

GET
Return a MultiDict containing all the variables from the QUERY_STRING.

POST
Return a MultiDict containing all the variables from a form request. Returns an empty dict-like object for non-form requests.

Form requests are typically POST requests, however PUT & PATCH requests with an appropriate Content-Type are also supported.

ResponseClass
alias of Response

accept
Gets and sets the Accept header (HTTP spec section 14.1).

accept_charset
Gets and sets the Accept-Charset header (HTTP spec section 14.2).

accept_encoding
Gets and sets the Accept-Encoding header (HTTP spec section 14.3).

accept_language
Gets and sets the Accept-Language header (HTTP spec section 14.4).

application_url
The URL including SCRIPT_NAME (no PATH_INFO or query string)

as_bytes(*skip_body=False*)
Return HTTP bytes representing this request. If skip_body is True, exclude the body. If skip_body is an integer larger than one, skip body only if its length is bigger than that number.

authorization
Gets and sets the `Authorization` header (HTTP spec section 14.8). Converts it using `parse_auth` and `serialize_auth`.

classmethod blank(*path*, *environ=None*, *base_url=None*, *headers=None*, *POST=None*, ***kw*)
Create a blank request environ (and Request wrapper) with the given path (path should be urlencoded), and any keys from environ.

The path will become path_info, with any query string split off and used.

All necessary keys will be added to the environ, but the values you pass in will take precedence. If you pass in base_url then wsgi.url_scheme, HTTP_HOST, and SCRIPT_NAME will be filled in from that value.

Any extra keyword will be passed to `__init__`.

body
Return the content of the request body.

body_file
Input stream of the request (wsgi.input). Setting this property resets the content_length and seekable flag (unlike setting req.body_file_raw).

body_file_raw
Gets and sets the `wsgi.input` key in the environment.

body_file_seekable
Get the body of the request (wsgi.input) as a seekable file-like object. Middleware and routing applications should use this attribute over .body_file.

If you access this value, CONTENT_LENGTH will also be updated.

cache_control
Get/set/modify the Cache-Control header (HTTP spec section 14.9)

call_application(*application*, *catch_exc_info=False*)
Call the given WSGI application, returning `(status_string, headerlist, app_iter)`

Be sure to call `app_iter.close()` if it's there.

If catch_exc_info is true, then returns `(status_string, headerlist, app_iter, exc_info)`, where the fourth item may be None, but won't be if there was an exception. If you don't do this and there was an exception, the exception will be raised directly.

client_addr
The effective client IP address as a string. If the `HTTP_X_FORWARDED_FOR` header exists in the WSGI environ, this attribute returns the client IP address present in that header (e.g. if the header value is `192.168.1.1, 192.168.1.2`, the value will be `192.168.1.1`). If no `HTTP_X_FORWARDED_FOR` header is present in the environ at all, this attribute will return the value of the `REMOTE_ADDR` header. If the `REMOTE_ADDR` header is unset, this attribute will return the value `None`.

> **Warning:** It is possible for user agents to put someone else's IP or just any string in `HTTP_X_FORWARDED_FOR` as it is a normal HTTP header. Forward proxies can also provide incorrect values (private IP addresses etc). You cannot "blindly" trust the result of this method to provide you with valid data unless you're certain that `HTTP_X_FORWARDED_FOR` has the correct values. The WSGI server must be behind a trusted proxy for this to be true.

content_length
Gets and sets the `Content-Length` header (HTTP spec section 14.13). Converts it using int.

content_type
Return the content type, but leaving off any parameters (like charset, but also things like the type in `application/atom+xml; type=entry`)

If you set this property, you can include parameters, or if you don't include any parameters in the value then existing parameters will be preserved.

cookies
Return a dictionary of cookies as found in the request.

copy()
Copy the request and environment object.

This only does a shallow copy, except of wsgi.input

copy_body()
Copies the body, in cases where it might be shared with another request object and that is not desired.

This copies the body in-place, either into a BytesIO object or a temporary file.

copy_get()
Copies the request and environment object, but turning this request into a GET along the way. If this was a POST request (or any other verb) then it becomes GET, and the request body is thrown away.

date
Gets and sets the `Date` header (HTTP spec section 14.8). Converts it using HTTP date.

domain
Returns the domain portion of the host value. Equivalent to:

```
domain = request.host
if ':' in domain:
    domain = domain.split(':', 1)[0]
```

This will be equivalent to the domain portion of the `HTTP_HOST` value in the environment if it exists, or the `SERVER_NAME` value in the environment if it doesn't. For example, if the environment contains an `HTTP_HOST` value of `foo.example.com:8000`, `request.domain` will return `foo.example.com`.

Note that this value cannot be *set* on the request. To set the host value use `webob.request.Request.host()` instead.

classmethod from_bytes(*b*)
Create a request from HTTP bytes data. If the bytes contain extra data after the request, raise a ValueError.

classmethod from_file(*fp*)
Read a request from a file-like object (it must implement `.read(size)` and `.readline()`).

It will read up to the end of the request, not the end of the file (unless the request is a POST or PUT and has no Content-Length, in that case, the entire file is read).

This reads the request as represented by `str(req)`; it may not read every valid HTTP request properly.

get_response(*application=None*, *catch_exc_info=False*)
Like `.call_application(application)`, except returns a response object with `.status`, `.headers`, and `.body` attributes.

This will use `self.ResponseClass` to figure out the class of the response object to return.

If `application` is not given, this will send the request to `self.make_default_send_app()`

headers
All the request headers as a case-insensitive dictionary-like object.

host
Host name provided in HTTP_HOST, with fall-back to SERVER_NAME

host_port
The effective server port number as a string. If the `HTTP_HOST` header exists in the WSGI environ, this attribute returns the port number present in that header. If the `HTTP_HOST` header exists but contains no explicit port number: if the WSGI url scheme is "https" , this attribute returns "443", if the WSGI url scheme is "http", this attribute returns "80" . If no `HTTP_HOST` header is present in the environ at all, this attribute will return the value of the `SERVER_PORT` header (which is guaranteed to be present).

host_url
The URL through the host (no path)

http_version
Gets and sets the `SERVER_PROTOCOL` key in the environment.

if_match
Gets and sets the `If-Match` header (HTTP spec section 14.24). Converts it as a Etag.

if_modified_since
Gets and sets the `If-Modified-Since` header (HTTP spec section 14.25). Converts it using HTTP date.

if_none_match
Gets and sets the `If-None-Match` header (HTTP spec section 14.26). Converts it as a Etag.

if_range
Gets and sets the `If-Range` header (HTTP spec section 14.27). Converts it using IfRange object.

if_unmodified_since
Gets and sets the `If-Unmodified-Since` header (HTTP spec section 14.28). Converts it using HTTP date.

is_body_readable
webob.is_body_readable is a flag that tells us that we can read the input stream even though CONTENT_LENGTH is missing. This allows FakeCGIBody to work and can be used by servers to support chunked encoding in requests. For background see https://bitbucket.org/ianb/webob/issue/6

is_body_seekable
Gets and sets the `webob.is_body_seekable` key in the environment.

is_xhr
Is X-Requested-With header present and equal to `XMLHttpRequest`?

Note: this isn't set by every XMLHttpRequest request, it is only set if you are using a Javascript library that sets it (or you set the header yourself manually). Currently Prototype and jQuery are known to set this header.

json
Access the body of the request as JSON

json_body
Access the body of the request as JSON

make_body_seekable()
This forces `environ['wsgi.input']` to be seekable. That means that, the content is copied into a BytesIO or temporary file and flagged as seekable, so that it will not be unnecessarily copied again.

After calling this method the .body_file is always seeked to the start of file and .content_length is not None.

The choice to copy to BytesIO is made from `self.request_body_tempfile_limit`

make_tempfile()
Create a tempfile to store big request body. This API is not stable yet. A 'size' argument might be added.

max_forwards
Gets and sets the `Max-Forwards` header (HTTP spec section 14.31). Converts it using int.

method
Gets and sets the `REQUEST_METHOD` key in the environment.

params
A dictionary-like object containing both the parameters from the query string and request body.

path
The path of the request, without host or query string

path_info
Gets and sets the `PATH_INFO` key in the environment.

path_info_peek()
Returns the next segment on PATH_INFO, or None if there is no next segment. Doesn't modify the environment.

path_info_pop(*pattern=None*)
'Pops' off the next segment of PATH_INFO, pushing it onto SCRIPT_NAME, and returning the popped segment. Returns None if there is nothing left on PATH_INFO.

Does not return `''` when there's an empty segment (like `/path//path`); these segments are just ignored.

Optional `pattern` argument is a regexp to match the return value before returning. If there is no match, no changes are made to the request and None is returned.

path_qs
The path of the request, without host but with query string

path_url
The URL including SCRIPT_NAME and PATH_INFO, but not QUERY_STRING

pragma
Gets and sets the `Pragma` header (HTTP spec section 14.32).

query_string
Gets and sets the `QUERY_STRING` key in the environment.

range
Gets and sets the `Range` header (HTTP spec section 14.35). Converts it using Range object.

referer
Gets and sets the `Referer` header (HTTP spec section 14.36).

referrer
Gets and sets the `Referer` header (HTTP spec section 14.36).

relative_url(*other_url*, *to_application=False*)
Resolve other_url relative to the request URL.

If `to_application` is True, then resolve it relative to the URL with only SCRIPT_NAME

remote_addr
Gets and sets the `REMOTE_ADDR` key in the environment.

remote_user
Gets and sets the `REMOTE_USER` key in the environment.

remove_conditional_headers(*remove_encoding=True*, *remove_range=True*, *remove_match=True*, *remove_modified=True*)
Remove headers that make the request conditional.

These headers can cause the response to be 304 Not Modified, which in some cases you may not want to be possible.

This does not remove headers like If-Match, which are used for conflict detection.

scheme
Gets and sets the `wsgi.url_scheme` key in the environment.

script_name
Gets and sets the `SCRIPT_NAME` key in the environment.

send(*application=None*, *catch_exc_info=False*)
Like `.call_application(application)`, except returns a response object with `.status`, `.headers`, and `.body` attributes.

This will use `self.ResponseClass` to figure out the class of the response object to return.

If `application` is not given, this will send the request to `self.make_default_send_app()`

server_name
Gets and sets the `SERVER_NAME` key in the environment.

server_port
Gets and sets the `SERVER_PORT` key in the environment. Converts it using int.

str_GET
<Deprecated attribute str_GET>

str_POST
<Deprecated attribute str_POST>

str_cookies
<Deprecated attribute str_cookies>

str_params
<Deprecated attribute str_params>

text
Get/set the text value of the body

upath_info
Gets and sets the `PATH_INFO` key in the environment.

url
The full request URL, including QUERY_STRING

url_encoding
Gets and sets the `webob.url_encoding` key in the environment.

urlargs
Return any *positional* variables matched in the URL.

Takes values from `environ['wsgiorg.routing_args']`. Systems like `routes` set this value.

urlvars
Return any *named* variables matched in the URL.

Takes values from `environ['wsgiorg.routing_args']`. Systems like `routes` set this value.

uscript_name
Gets and sets the `SCRIPT_NAME` key in the environment.

user_agent
Gets and sets the `User-Agent` header (HTTP spec section 14.43).

4.7 `webob.response` – Response

4.7.1 Response

class `webob.response.`**Response**(*body=None*, *status=None*, *headerlist=None*, *app_iter=None*, *content_type=None*, *conditional_response=None*, ***kw*)
Represents a WSGI response

accept_ranges
Gets and sets the `Accept-Ranges` header (HTTP spec section 14.5).

age
Gets and sets the `Age` header (HTTP spec section 14.6). Converts it using int.

allow
Gets and sets the `Allow` header (HTTP spec section 14.7). Converts it using list.

app_iter
Returns the app_iter of the response.

If body was set, this will create an app_iter from that body (a single-item list)

app_iter_range(*start*, *stop*)
Return a new app_iter built from the response app_iter, that serves up only the given `start:stop` range.

body
The body of the response, as a `str`. This will read in the entire app_iter if necessary.

body_file
A file-like object that can be used to write to the body. If you passed in a list app_iter, that app_iter will be modified by writes.

cache_control
Get/set/modify the Cache-Control header (HTTP spec section 14.9)

charset
Get/set the charset (in the Content-Type)

conditional_response_app(*environ*, *start_response*)
Like the normal __call__ interface, but checks conditional headers:

- If-Modified-Since (304 Not Modified; only on GET, HEAD)
- If-None-Match (304 Not Modified; only on GET, HEAD)
- Range (406 Partial Content; only on GET, HEAD)

content_disposition
Gets and sets the `Content-Disposition` header (HTTP spec section 19.5.1).

content_encoding
Gets and sets the `Content-Encoding` header (HTTP spec section 14.11).

content_language
Gets and sets the `Content-Language` header (HTTP spec section 14.12). Converts it using list.

content_length
Gets and sets the `Content-Length` header (HTTP spec section 14.17). Converts it using int.

content_location
Gets and sets the `Content-Location` header (HTTP spec section 14.14).

content_md5
Gets and sets the `Content-MD5` header (HTTP spec section 14.14).

content_range
Gets and sets the `Content-Range` header (HTTP spec section 14.16). Converts it using ContentRange object.

content_type
Get/set the Content-Type header (or None), *without* the charset or any parameters.

If you include parameters (or `;` at all) when setting the content_type, any existing parameters will be deleted; otherwise they will be preserved.

content_type_params
A dictionary of all the parameters in the content type.

(This is not a view, set to change, modifications of the dict would not be applied otherwise)

copy()
Makes a copy of the response

date
Gets and sets the `Date` header (HTTP spec section 14.18). Converts it using HTTP date.

delete_cookie(*name*, *path='/'*, *domain=None*)
Delete a cookie from the client. Note that path and domain must match how the cookie was originally set.

This sets the cookie to the empty string, and max_age=0 so that it should expire immediately.

encode_content(*encoding='gzip'*, *lazy=False*)
Encode the content with the given encoding (only gzip and identity are supported).

etag
Gets and sets the `ETag` header (HTTP spec section 14.19). Converts it using Entity tag.

expires
Gets and sets the `Expires` header (HTTP spec section 14.21). Converts it using HTTP date.

classmethod from_file(*fp*)
Reads a response from a file-like object (it must implement `.read(size)` and `.readline()`).

It will read up to the end of the response, not the end of the file.

This reads the response as represented by `str(resp)`; it may not read every valid HTTP response properly. Responses must have a `Content-Length`

headerlist
The list of response headers

headers
The headers in a dictionary-like object

json
Access the body of the response as JSON

json_body
Access the body of the response as JSON

last_modified
Gets and sets the `Last-Modified` header (HTTP spec section 14.29). Converts it using HTTP date.

location
Gets and sets the `Location` header (HTTP spec section 14.30).

md5_etag(*body=None*, *set_content_md5=False*)
Generate an etag for the response object using an MD5 hash of the body (the body parameter, or `self.body` if not given)

Sets `self.etag` If `set_content_md5` is True sets `self.content_md5` as well

merge_cookies(*resp*)
Merge the cookies that were set on this response with the given *resp* object (which can be any WSGI application).

If the *resp* is a `webob.Response` object, then the other object will be modified in-place.

pragma
Gets and sets the `Pragma` header (HTTP spec section 14.32).

retry_after
Gets and sets the `Retry-After` header (HTTP spec section 14.37). Converts it using HTTP date or delta seconds.

server
Gets and sets the `Server` header (HTTP spec section 14.38).

set_cookie(*name=None*, *value=''*, *max_age=None*, *path='/'*, *domain=None*, *secure=False*, *httponly=False*, *comment=None*, *expires=None*, *overwrite=False*, *key=None*)
Set (add) a cookie for the response.

Arguments are:

`name`

The cookie name.

`value`

The cookie value, which should be a string or `None`. If `value` is `None`, it's equivalent to calling the *`webob.response.Response.unset_cookie()`* method for this cookie key (it effectively deletes the cookie on the client).

`max_age`

An integer representing a number of seconds, `datetime.timedelta`, or `None`. This value is used as the `Max-Age` of the generated cookie. If `expires` is not passed and this value is not `None`, the `max_age` value will also influence the `Expires` value of the cookie (`Expires` will be set to now + max_age). If this value is `None`, the cookie will not have a `Max-Age` value (unless `expires` is set). If both `max_age` and `expires` are set, this value takes precedence.

`path`

A string representing the cookie `Path` value. It defaults to `/`.

`domain`

A string representing the cookie `Domain`, or `None`. If domain is `None`, no `Domain` value will be sent in the cookie.

`secure`

A boolean. If it's `True`, the `secure` flag will be sent in the cookie, if it's `False`, the `secure` flag will not be sent in the cookie.

`httponly`

A boolean. If it's `True`, the `HttpOnly` flag will be sent in the cookie, if it's `False`, the `HttpOnly` flag will not be sent in the cookie.

`comment`

A string representing the cookie `Comment` value, or `None`. If `comment` is `None`, no `Comment` value will be sent in the cookie.

`expires`

A `datetime.timedelta` object representing an amount of time, `datetime.datetime` or `None`. A non-`None` value is used to generate the `Expires` value of the generated cookie. If `max_age` is not passed, but this value is not `None`, it will influence the `Max-Age` header. If this value is `None`, the `Expires` cookie value will be unset (unless `max_age` is set). If `max_age` is set, it will be used to generate the `expires` and this value is ignored.

`overwrite`

If this key is `True`, before setting the cookie, unset any existing cookie.

status
The status string

status_code
The status as an integer

status_int
The status as an integer

text
Get/set the text value of the body (using the charset of the Content-Type)

ubody
Deprecated alias for .text

unicode_body
Deprecated alias for .text

unset_cookie(*name*, *strict=True*)
Unset a cookie with the given name (remove it from the response).

vary
Gets and sets the `Vary` header (HTTP spec section 14.44). Converts it using list.

www_authenticate
Gets and sets the `WWW-Authenticate` header (HTTP spec section 14.47). Converts it using `parse_auth` and `serialize_auth`.

class `webob.response.`**AppIterRange**(*app_iter*, *start*, *stop*)
Wraps an app_iter, returning just a range of bytes

4.8 webob.static – Serving static files

class webob.static.**FileApp**(*filename*, ***kw*)
An application that will send the file at the given filename.

Adds a mime type based on *mimetypes.guess_type()*.

class webob.static.**DirectoryApp**(*path*, *index_page='index.html'*, *hide_index_with_redirect=False*, ***kw*)
An application that serves up the files in a given directory.

This will serve index files (by default `index.html`), or set `index_page=None` to disable this. If you set `hide_index_with_redirect=True` (it defaults to False) then requests to, e.g., `/index.html` will be redirected to `/`.

To customize *FileApp* instances creation (which is what actually serves the responses), override the *make_fileapp* method.

4.9 webob – Request/Response objects

4.9.1 Headers

Accept-*

Parses a variety of `Accept-*` headers.

These headers generally take the form of:

```
value1; q=0.5, value2; q=0
```

Where the `q` parameter is optional. In theory other parameters exists, but this ignores them.

class webob.acceptparse.**Accept**(*header_value*)
Represents a generic `Accept-*` style header.

This object should not be modified. To add items you can use `accept_obj + 'accept_thing'` to get a new object

best_match(*offers*, *default_match=None*)
Returns the best match in the sequence of offered types.

The sequence can be a simple sequence, or you can have `(match, server_quality)` items in the sequence. If you have these tuples then the client quality is multiplied by the server_quality to get a total. If two matches have equal weight, then the one that shows up first in the *offers* list will be returned.

But among matches with the same quality the match to a more specific requested type will be chosen. For example a match to text/* trumps */*.

default_match (default None) is returned if there is no intersection.

first_match(*offers*)
DEPRECATED Returns the first allowed offered type. Ignores quality. Returns the first offered type if nothing else matches; or if you include None at the end of the match list then that will be returned.

static parse(*value*)
Parse `Accept-*` style header.

Return iterator of `(value, quality)` pairs. `quality` defaults to 1.

quality(*offer*, *modifier=1*)
Return the quality of the given offer. Returns None if there is no match (not 0).

class webob.acceptparse.**MIMEAccept**(*header_value*)
Represents the Accept header, which is a list of mimetypes.

This class knows about mime wildcards, like image/*

accept_html()
Returns true if any HTML-like type is accepted

accepts_html
Returns true if any HTML-like type is accepted

Cache-Control

class webob.cachecontrol.**CacheControl**(*properties*, *type*)
Represents the Cache-Control header.

By giving a type of 'request' or 'response' you can control what attributes are allowed (some Cache-Control values only apply to requests or responses).

copy()
Returns a copy of this object.

classmethod parse(*header*, *updates_to=None*, *type=None*)
Parse the header, returning a CacheControl object.

The object is bound to the request or response object updates_to, if that is given.

update_dict
alias of UpdateDict

Range and related headers

class webob.byterange.**Range**(*start*, *end*)
Represents the Range header.

content_range(*length*)
Works like range_for_length; returns None or a ContentRange object

You can use it like:

```
response.content_range = req.range.content_range(response.content_length)
```

Though it's still up to you to actually serve that content range!

classmethod parse(*header*)
Parse the header; may return None if header is invalid

range_for_length(*length*)
If there is only one range, and *if* it is satisfiable by the given length, then return a (start, end) non-inclusive range of bytes to serve. Otherwise return None

class webob.byterange.**ContentRange**(*start*, *stop*, *length*)
Represents the Content-Range header

This header is start-stop/length, where start-stop and length can be * (represented as None in the attributes).

classmethod **parse**(*value*)
Parse the header. May return None if it cannot parse.

class webob.etag.**IfRange**(*etag*)

classmethod **parse**(*value*)
Parse this from a header value.

ETag

class webob.etag.**ETagMatcher**(*etags*)

classmethod **parse**(*value*, *strong=True*)
Parse this from a header value

4.9.2 Misc Functions and Internals

webob.**html_escape**(*s*)
HTML-escape a string or object

This converts any non-string objects passed into it to strings (actually, using `unicode()`). All values returned are non-unicode strings (using `&#num;` entities for all non-ASCII characters).

None is treated specially, and returns the empty string.

class webob.headers.**ResponseHeaders**(**args*, ***kw*)
Dictionary view on the response headerlist. Keys are normalized for case and whitespace.

class webob.headers.**EnvironHeaders**(*environ*)
An object that represents the headers as present in a WSGI environment.

This object is a wrapper (with no internal state) for a WSGI request object, representing the CGI-style HTTP_* keys as a dictionary. Because a CGI environment can only hold one value for each key, this dictionary is single-valued (unlike outgoing headers).

class webob.cachecontrol.**UpdateDict**
Dict that has a callback on all updates

CHAPTER 5

Request

The request object is a wrapper around the WSGI environ dictionary. This dictionary contains keys for each header, keys that describe the request (including the path and query string), a file-like object for the request body, and a variety of custom keys. You can always access the environ with `req.environ`.

Some of the most important/interesting attributes of a request object:

`req.method`: The request method, e.g., `'GET'`, `'POST'`

`req.GET`: A *dictionary-like object* with all the variables in the query string.

`req.POST`: A *dictionary-like object* with all the variables in the request body. This only has variables if the request was a `POST` and it is a form submission.

`req.params`: A *dictionary-like object* with a combination of everything in `req.GET` and `req.POST`.

`req.body`: The contents of the body of the request. This contains the entire request body as a string. This is useful when the request is a `POST` that is *not* a form submission, or a request like a `PUT`. You can also get `req.body_file` for a file-like object.

`req.cookies`: A simple dictionary of all the cookies.

`req.headers`: A dictionary of all the headers. This is dictionary is case-insensitive.

`req.urlvars` and `req.urlargs`: `req.urlvars` is the keyword parameters associated with the request URL. `req.urlargs` are the positional parameters. These are set by products like Routes and Selector.

Also, for standard HTTP request headers there are usually attributes, for instance: `req.accept_language`, `req.content_length`, `req.user_agent`, as an example. These properties expose the *parsed* form of each header, for whatever parsing makes sense. For instance, `req.if_modified_since` returns a datetime object (or None if the header is was not provided). Details are in the Request reference.

5.1 URLs

In addition to these attributes, there are several ways to get the URL of the request. I'll show various values for an example URL `http://localhost/app-root/doc?article_id=10`, where the application is mounted at `http://localhost/app-root`.

`req.url`: The full request URL, with query string, e.g., `'http://localhost/app-root/doc?article_id=10'`

`req.application_url`: The URL of the application (just the SCRIPT_NAME portion of the path, not PATH_INFO). E.g., `'http://localhost/app-root'`

`req.host_url`: The URL with the host, e.g., `'http://localhost'`

req.relative_url(url, to_application=False): Gives a URL, relative to the current URL. If `to_application` is True, then resolves it relative to `req.application_url`.

5.2 Methods

There are several methods in `webob.Request` but only a few you'll use often:

Request.blank(base_url): Creates a new request with blank information, based at the given URL. This can be useful for subrequests and artificial requests. You can also use `req.copy()` to copy an existing request, or for subrequests `req.copy_get()` which copies the request but always turns it into a GET (which is safer to share for subrequests).

req.get_response(wsgi_application): This method calls the given WSGI application with this request, and returns a *Response* object. You can also use this for subrequests or testing.

5.3 Unicode

Many of the properties in the request object will return unicode values if the request encoding/charset is provided. The client *can* indicate the charset with something like `Content-Type: application/x-www-form-urlencoded; charset=utf8`, but browsers seldom set this. You can set the charset with `req.charset = 'utf8'`, or during instantiation with `Request(environ, charset='utf8')`. If you subclass `Request` you can also set `charset` as a class-level attribute.

If it is set, then `req.POST`, `req.GET`, `req.params`, and `req.cookies` will contain unicode strings. Each has a corresponding `req.str_*` (like `req.str_POST`) that is always `str` and never unicode.

CHAPTER 6

Response

The response object looks a lot like the request object, though with some differences. The request object wraps a single `environ` object; the response object has three fundamental parts (based on WSGI):

`response.status`: The response code plus message, like `'200 OK'`. To set the code without the reason, use `response.status_code = 200`.

`response.headerlist`: A list of all the headers, like `[('Content-Type', 'text/html')]`. There's a case-insensitive *dictionary-like object* in `response.headers` that also allows you to access these same headers.

`response.app_iter`: An iterable (such as a list or generator) that will produce the content of the response. This is also accessible as `response.body` (a string), `response.unicode_body` (a unicode object, informed by `response.charset`), and `response.body_file` (a file-like object; writing to it appends to `app_iter`).

Everything else in the object derives from this underlying state. Here's the highlights:

`response.content_type`: The content type *not* including the `charset` parameter. Typical use: `response.content_type = 'text/html'`. You can subclass `Response` and add a class-level attribute `default_content_type` to set this automatically on instantiation.

`response.charset`: The `charset` parameter of the content-type, it also informs encoding in `response.unicode_body`. `response.content_type_params` is a dictionary of all the parameters.

`response.request`: This optional attribute can point to the request object associated with this response object.

`response.set_cookie(key, value, max_age=None, path='/', domain=None, secure=None, httponly=` Set a cookie. The keyword arguments control the various cookie parameters. The `max_age` argument is the length for the cookie to live in seconds (you may also use a timedelta object). The *Expires*' key will also be set based on the value of `max_age`.

`response.delete_cookie(key, path='/', domain=None):` Delete a cookie from the client. This sets `max_age` to 0 and the cookie value to `''`.

`response.cache_expires(seconds=0):` This makes this response cachable for the given number of seconds, or if `seconds` is 0 then the response is uncacheable (this also sets the `Expires` header).

`response(environ, start_response):` The response object is a WSGI application. As an application, it acts according to how you create it. It *can* do conditional responses if you pass `conditional_response=True` when instantiating (or set that attribute later). It can also do HEAD and Range requests.

6.1 Headers

Like the request, most HTTP response headers are available as properties. These are parsed, so you can do things like `response.last_modified = os.path.getmtime(filename)`.

See also:

webob.response.Response

6.2 Instantiating the Response

Of course most of the time you just want to *make* a response. Generally any attribute of the response can be passed in as a keyword argument to the class; e.g.:

```
response = Response(body='hello world!', content_type='text/plain')
```

The status defaults to `'200 OK'`. The content_type does not default to anything, though if you subclass `Response` and set `default_content_type` you can override this behavior.

CHAPTER 7

Exceptions

To facilitate error responses like 404 Not Found, the module `webob.exc` contains classes for each kind of error response. These include boring but appropriate error bodies.

Each class is named `webob.exc.HTTP*`, where `*` is the reason for the error. For instance, `webob.exc.HTTPNotFound`. It subclasses `Response`, so you can manipulate the instances in the same way. A typical example is:

```
response = HTTPNotFound('There is no such resource')
# or:
response = HTTPMovedPermanently(location=new_url)
```

You can use this like:

```
try:
    ... stuff ...
    raise HTTPNotFound('No such resource')
except HTTPException, e:
    return e(environ, start_response)
```

The exceptions are still WSGI applications, but you cannot set attributes like `content_type`, `charset`, etc. on these exception objects.

CHAPTER 8

Multidict

Several parts of WebOb use a "multidict"; this is a dictionary where a key can have multiple values. The quintessential example is a query string like `?pref=red&pref=blue`; the `pref` variable has two values: `red` and `blue`.

In a multidict, when you do `request.GET['pref']` you'll get back only `'blue'` (the last value of `pref`). Sometimes returning a string, and sometimes returning a list, is the cause of frequent exceptions. If you want *all* the values back, use `request.GET.getall('pref')`. If you want to be sure there is *one and only one* value, use `request.GET.getone('pref')`, which will raise an exception if there is zero or more than one value for `pref`.

When you use operations like `request.GET.items()` you'll get back something like `[('pref', 'red'), ('pref', 'blue')]`. All the key/value pairs will show up. Similarly `request.GET.keys()` returns `['pref', 'pref']`. Multidict is a view on a list of tuples; all the keys are ordered, and all the values are ordered.

CHAPTER 9

Example

The file-serving example shows how to do more advanced HTTP techniques, while the comment middleware example shows middleware. For applications it's more reasonable to use WebOb in the context of a larger framework. Pylons uses WebOb in 0.9.7+.

9.1 WebOb File-Serving Example

This document shows how you can make a static-file-serving application using WebOb. We'll quickly build this up from minimal functionality to a high-quality file serving application.

Note: Starting from 1.2b4, WebOb ships with a *webob.static* module which implements a *webob.static.FileApp* WSGI application similar to the one described below.

This document stays as a didactic example how to serve files with WebOb, but you should consider using applications from *webob.static* in production.

First we'll setup a really simple shim around our application, which we can use as we improve our application:

```
>>> from webob import Request, Response
>>> import os
>>> class FileApp(object):
...     def __init__(self, filename):
...         self.filename = filename
...     def __call__(self, environ, start_response):
...         res = make_response(self.filename)
...         return res(environ, start_response)
>>> import mimetypes
>>> def get_mimetype(filename):
...     type, encoding = mimetypes.guess_type(filename)
...     # We'll ignore encoding, even though we shouldn't really
...     return type or 'application/octet-stream'
```

Now we can make different definitions of `make_response`. The simplest version:

```
>>> def make_response(filename):
...     res = Response(content_type=get_mimetype(filename))
...     res.body = open(filename, 'rb').read()
...     return res
```

We'll test it out with a file `test-file.txt` in the WebOb doc directory, which has the following content:

```
This is a test.  Hello test people!
```

Let's give it a shot:

```
>>> fn = os.path.join(doc_dir, 'file-example-code/test-file.txt')
>>> open(fn).read()
'This is a test.  Hello test people!'
>>> app = FileApp(fn)
>>> req = Request.blank('/')
>>> print req.get_response(app)
200 OK
Content-Type: text/plain; charset=UTF-8
Content-Length: 35

This is a test.  Hello test people!
```

Well, that worked. But it's not a very fancy object. First, it reads everything into memory, and that's bad. We'll create an iterator instead:

```
>>> class FileIterable(object):
...     def __init__(self, filename):
...         self.filename = filename
...     def __iter__(self):
...         return FileIterator(self.filename)
>>> class FileIterator(object):
...     chunk_size = 4096
...     def __init__(self, filename):
...         self.filename = filename
...         self.fileobj = open(self.filename, 'rb')
...     def __iter__(self):
...         return self
...     def next(self):
...         chunk = self.fileobj.read(self.chunk_size)
...         if not chunk:
...             raise StopIteration
...         return chunk
...     __next__ = next # py3 compat
>>> def make_response(filename):
...     res = Response(content_type=get_mimetype(filename))
...     res.app_iter = FileIterable(filename)
...     res.content_length = os.path.getsize(filename)
...     return res
```

And testing:

```
>>> req = Request.blank('/')
>>> print req.get_response(app)
200 OK
Content-Type: text/plain; charset=UTF-8
Content-Length: 35

This is a test.  Hello test people!
```

Well, that doesn't *look* different, but lets *imagine* that it's different because we know we changed some code. Now to add some basic metadata to the response:

```
>>> def make_response(filename):
...     res = Response(content_type=get_mimetype(filename),
...                    conditional_response=True)
```

```
...     res.app_iter = FileIterable(filename)
...     res.content_length = os.path.getsize(filename)
...     res.last_modified = os.path.getmtime(filename)
...     res.etag = '%s-%s-%s' % (os.path.getmtime(filename),
...                              os.path.getsize(filename), hash(filename))
...     return res
```

Now, with `conditional_response` on, and with `last_modified` and `etag` set, we can do conditional requests:

```
>>> req = Request.blank('/')
>>> res = req.get_response(app)
>>> print res
200 OK
Content-Type: text/plain; charset=UTF-8
Content-Length: 35
Last-Modified: ... GMT
ETag: ...-...

This is a test.  Hello test people!
>>> req2 = Request.blank('/')
>>> req2.if_none_match = res.etag
>>> req2.get_response(app)
<Response ... 304 Not Modified>
>>> req3 = Request.blank('/')
>>> req3.if_modified_since = res.last_modified
>>> req3.get_response(app)
<Response ... 304 Not Modified>
```

We can even do Range requests, but it will currently involve iterating through the file unnecessarily. When there's a range request (and you set `conditional_response=True`) the application will satisfy that request. But with an arbitrary iterator the only way to do that is to run through the beginning of the iterator until you get to the chunk that the client asked for. We can do better because we can use `fileobj.seek(pos)` to move around the file much more efficiently.

So we'll add an extra method, `app_iter_range`, that `Response` looks for:

```
>>> class FileIterable(object):
...     def __init__(self, filename, start=None, stop=None):
...         self.filename = filename
...         self.start = start
...         self.stop = stop
...     def __iter__(self):
...         return FileIterator(self.filename, self.start, self.stop)
...     def app_iter_range(self, start, stop):
...         return self.__class__(self.filename, start, stop)
>>> class FileIterator(object):
...     chunk_size = 4096
...     def __init__(self, filename, start, stop):
...         self.filename = filename
...         self.fileobj = open(self.filename, 'rb')
...         if start:
...             self.fileobj.seek(start)
...         if stop is not None:
...             self.length = stop - start
...         else:
...             self.length = None
...     def __iter__(self):
...         return self
```

```
...     def next(self):
...         if self.length is not None and self.length <= 0:
...             raise StopIteration
...         chunk = self.fileobj.read(self.chunk_size)
...         if not chunk:
...             raise StopIteration
...         if self.length is not None:
...             self.length -= len(chunk)
...             if self.length < 0:
...                 # Chop off the extra:
...                 chunk = chunk[:self.length]
...         return chunk
...     __next__ = next # py3 compat
```

Now we'll test it out:

```
>>> req = Request.blank('/')
>>> res = req.get_response(app)
>>> req2 = Request.blank('/')
>>> # Re-fetch the first 5 bytes:
>>> req2.range = (0, 5)
>>> res2 = req2.get_response(app)
>>> res2
<Response ... 206 Partial Content>
>>> # Let's check it's our custom class:
>>> res2.app_iter
<FileIterable object at ...>
>>> res2.body
'This '
>>> # Now, conditional range support:
>>> req3 = Request.blank('/')
>>> req3.if_range = res.etag
>>> req3.range = (0, 5)
>>> req3.get_response(app)
<Response ... 206 Partial Content>
>>> req3.if_range = 'invalid-etag'
>>> req3.get_response(app)
<Response ... 200 OK>
```

9.2 Wiki Example

author Ian Bicking <ianb@colorstudy.com>

Contents

9.2.1 Introduction

This is an example of how to write a WSGI application using WebOb. WebOb isn't itself intended to write applications – it is not a web framework on its own – but it is *possible* to write applications using just WebOb.

The file serving example is a better example of advanced HTTP usage. The comment middleware example is a better example of using middleware. This example provides some completeness by showing an application-focused end point.

This example implements a very simple wiki.

9.2.2 Code

The finished code for this is available in docs/wiki-example-code/example.py – you can run that file as a script to try it out.

9.2.3 Creating an Application

A common pattern for creating small WSGI applications is to have a class which is instantiated with the configuration. For our application we'll be storing the pages under a directory.

```
class WikiApp(object):

    def __init__(self, storage_dir):
        self.storage_dir = os.path.abspath(os.path.normpath(storage_dir))
```

WSGI applications are callables like `wsgi_app(environ, start_response)`. *Instances* of *WikiApp* are WSGI applications, so we'll implement a `__call__` method:

```
class WikiApp(object):
    ...
    def __call__(self, environ, start_response):
        # what we'll fill in
```

To make the script runnable we'll create a simple command-line interface:

```
if __name__ == '__main__':
    import optparse
    parser = optparse.OptionParser(
```

```
        usage='%prog --port=PORT'
        )
    parser.add_option(
        '-p', '--port',
        default='8080',
        dest='port',
        type='int',
        help='Port to serve on (default 8080)')
    parser.add_option(
        '--wiki-data',
        default='./wiki',
        dest='wiki_data',
        help='Place to put wiki data into (default ./wiki/)')
    options, args = parser.parse_args()
    print 'Writing wiki pages to %s' % options.wiki_data
    app = WikiApp(options.wiki_data)
    from wsgiref.simple_server import make_server
    httpd = make_server('localhost', options.port, app)
    print 'Serving on http://localhost:%s' % options.port
    try:
        httpd.serve_forever()
    except KeyboardInterrupt:
        print '^C'
```

There's not much to talk about in this code block. The application is instantiated and served with the built-in module wsgiref.simple_server.

9.2.4 The WSGI Application

Of course all the interesting stuff is in that `__call__` method. WebOb lets you ignore some of the details of WSGI, like what `start_response` really is. `environ` is a CGI-like dictionary, but `webob.Request` gives an object interface to it. `webob.Response` represents a response, and is itself a WSGI application. Here's kind of the hello world of WSGI applications using these objects:

```
from webob import Request, Response

class WikiApp(object):
    ...

    def __call__(self, environ, start_response):
        req = Request(environ)
        resp = Response(
            'Hello %s!' % req.params.get('name', 'World'))
        return resp(environ, start_response)
```

`req.params.get('name', 'World')` gets any query string parameter (like `?name=Bob`), or if it's a POST form request it will look for a form parameter `name`. We instantiate the response with the body of the response. You could also give keyword arguments like `content_type='text/plain'` (`text/html` is the default content type and `200 OK` is the default status).

For the wiki application we'll support a couple different kinds of screens, and we'll make our `__call__` method dispatch to different methods depending on the request. We'll support an `action` parameter like `?action=edit`, and also dispatch on the method (GET, POST, etc, in `req.method`). We'll pass in the request and expect a response object back.

Also, WebOb has a series of exceptions in `webob.exc`, like `webob.exc.HTTPNotFound`, `webob.exc.HTTPTemporaryRedirect`, etc. We'll also let the method raise one of these exceptions and

turn it into a response.

One last thing we'll do in our `__call__` method is create our `Page` object, which represents a wiki page.

All this together makes:

```
from webob import Request, Response
from webob import exc

class WikiApp(object):
    ...

    def __call__(self, environ, start_response):
        req = Request(environ)
        action = req.params.get('action', 'view')
        # Here's where we get the Page domain object:
        page = self.get_page(req.path_info)
        try:
            try:
                # The method name is action_{action_param}_{request_method}:
                meth = getattr(self, 'action_%s_%s' % (action, req.method))
            except AttributeError:
                # If the method wasn't found there must be
                # something wrong with the request:
                raise exc.HTTPBadRequest('No such action %r' % action)
            resp = meth(req, page)
        except exc.HTTPException, e:
            # The exception object itself is a WSGI application/response:
            resp = e
        return resp(environ, start_response)
```

9.2.5 The Domain Object

The `Page` domain object isn't really related to the web, but it is important to implementing this. Each `Page` is just a file on the filesystem. Our `get_page` method figures out the filename given the path (the path is in `req.path_info`, which is all the path after the base path). The `Page` class handles getting and setting the title and content.

Here's the method to figure out the filename:

```
import os

class WikiApp(object):
    ...

    def get_page(self, path):
        path = path.lstrip('/')
        if not path:
            # The path was '/', the home page
            path = 'index'
        path = os.path.join(self.storage_dir)
        path = os.path.normpath(path)
        if path.endswith('/'):
            path += 'index'
        if not path.startswith(self.storage_dir):
            raise exc.HTTPBadRequest("Bad path")
        path += '.html'
        return Page(path)
```

Mostly this is just the kind of careful path construction you have to do when mapping a URL to a filename. While the server *may* normalize the path (so that a path like `/../../` can't be requested), you can never really be sure. By using `os.path.normpath` we eliminate these, and then we make absolutely sure that the resulting path is under our `self.storage_dir` with `if not path.startswith(self.storage_dir): raise exc.HTTPBadRequest("Bad path")`.

Here's the actual domain object:

```
class Page(object):
    def __init__(self, filename):
        self.filename = filename

    @property
    def exists(self):
        return os.path.exists(self.filename)

    @property
    def title(self):
        if not self.exists:
            # we need to guess the title
            basename = os.path.splitext(os.path.basename(self.filename))[0]
            basename = re.sub(r'[_-]', ' ', basename)
            return basename.capitalize()
        content = self.full_content
        match = re.search(r'<title>(.*?)</title>', content, re.I|re.S)
        return match.group(1)

    @property
    def full_content(self):
        f = open(self.filename, 'rb')
        try:
            return f.read()
        finally:
            f.close()

    @property
    def content(self):
        if not self.exists:
            return ''
        content = self.full_content
        match = re.search(r'<body[^>]*>(.*?)</body>', content, re.I|re.S)
        return match.group(1)

    @property
    def mtime(self):
        if not self.exists:
            return None
        else:
            return int(os.stat(self.filename).st_mtime)

    def set(self, title, content):
        dir = os.path.dirname(self.filename)
        if not os.path.exists(dir):
            os.makedirs(dir)
        new_content = """<html><head><title>%s</title></head><body>%s</body></html>""" % (
            title, content)
        f = open(self.filename, 'wb')
        f.write(new_content)
        f.close()
```

Basically it provides a `.title` attribute, a `.content` attribute, the `.mtime` (last modified time), and the page can exist or not (giving appropriate guesses for title and content when the page does not exist). It encodes these on the filesystem as a simple HTML page that is parsed by some regular expressions.

None of this really applies much to the web or WebOb, so I'll leave it to you to figure out the details of this.

9.2.6 URLs, PATH_INFO, and SCRIPT_NAME

This is an aside for the tutorial, but an important concept. In WSGI, and accordingly with WebOb, the URL is split up into several pieces. Some of these are obvious and some not.

An example:

```
http://example.com:8080/wiki/article/12?version=10
```

There are several components here:

- req.scheme: `http`
- req.host: `example.com:8080`
- req.server_name: `example.com`
- req.server_port: 8080
- req.script_name: `/wiki`
- req.path_info: `/article/12`
- req.query_string: `version=10`

One non-obvious part is `req.script_name` and `req.path_info`. These correspond to the CGI environmental variables `SCRIPT_NAME` and `PATH_INFO`. `req.script_name` points to the *application*. You might have several applications in your site at different paths: one at `/wiki`, one at `/blog`, one at `/`. Each application doesn't necessarily know about the others, but it has to construct its URLs properly – so any internal links to the wiki application should start with `/wiki`.

Just as there are pieces to the URL, there are several properties in WebOb to construct URLs based on these:

- req.host_url: `http://example.com:8080`
- req.application_url: `http://example.com:8080/wiki`
- req.path_url: `http://example.com:8080/wiki/article/12`
- req.path: `/wiki/article/12`
- req.path_qs: `/wiki/article/12?version=10`
- req.url: `http://example.com:8080/wiki/article/12?version10`

You can also create URLs with `req.relative_url('some/other/page')`. In this example that would resolve to `http://example.com:8080/wiki/article/some/other/page`. You can also create a relative URL to the application URL (SCRIPT_NAME) like `req.relative_url('some/other/page', True)` which would be `http://example.com:8080/wiki/some/other/page`.

9.2.7 Back to the Application

We have a dispatching function with `__call__` and we have a domain object with `Page`, but we aren't actually doing anything.

The dispatching goes to `action_ACTION_METHOD`, where ACTION defaults to `view`. So a simple page view will be `action_view_GET`. Let's implement that:

```
class WikiApp(object):
    ...

    def action_view_GET(self, req, page):
        if not page.exists:
            return exc.HTTPTemporaryRedirect(
                location=req.url + '?action=edit')
        text = self.view_template.substitute(
            page=page, req=req)
        resp = Response(text)
        resp.last_modified = page.mtime
        resp.conditional_response = True
        return resp
```

The first thing we do is redirect the user to the edit screen if the page doesn't exist. `exc.HTTPTemporaryRedirect` is a response that gives a `307 Temporary Redirect` response with the given location.

Otherwise we fill in a template. The template language we're going to use in this example is Tempita, a very simple template language with a similar interface to string.Template.

The template actually looks like this:

```
from tempita import HTMLTemplate

VIEW_TEMPLATE = HTMLTemplate("""\
<html>
 <head>
  <title>{{page.title}}</title>
 </head>
 <body>
<h1>{{page.title}}</h1>

<div>{{page.content|html}}</div>

<hr>
<a href="{{req.url}}?action=edit">Edit</a>
 </body>
</html>
""")

class WikiApp(object):
    view_template = VIEW_TEMPLATE
    ...
```

As you can see it's a simple template using the title and the body, and a link to the edit screen. We copy the template object into a class method (`view_template = VIEW_TEMPLATE`) so that potentially a subclass could override these templates.

`tempita.HTMLTemplate` is a template that does automatic HTML escaping. Our wiki will just be written in plain HTML, so we disable escaping of the content with `{{page.content|html}}`.

So let's look at the `action_view_GET` method again:

```
def action_view_GET(self, req, page):
    if not page.exists:
        return exc.HTTPTemporaryRedirect(
```

```
            location=req.url + '?action=edit')
    text = self.view_template.substitute(
        page=page, req=req)
    resp = Response(text)
    resp.last_modified = page.mtime
    resp.conditional_response = True
    return resp
```

The template should be pretty obvious now. We create a response with `Response(text)`, which already has a default Content-Type of `text/html`.

To allow conditional responses we set `resp.last_modified`. You can set this attribute to a date, None (effectively removing the header), a time tuple (like produced by `time.localtime()`), or as in this case to an integer timestamp. If you get the value back it will always be a datetime object (or None). With this header we can process requests with If-Modified-Since headers, and return `304 Not Modified` if appropriate. It won't actually do that unless you set `resp.conditional_response` to True.

Note: If you subclass `webob.Response` you can set the class attribute `default_conditional_response = True` and this setting will be on by default. You can also set other defaults, like the `default_charset` (`"utf8"`), or `default_content_type` (`"text/html"`).

9.2.8 The Edit Screen

The edit screen will be implemented in the method `action_edit_GET`. There's a template and a very simple method:

```
EDIT_TEMPLATE = HTMLTemplate("""\
<html>
 <head>
  <title>Edit: {{page.title}}</title>
 </head>
 <body>
{{if page.exists}}
<h1>Edit: {{page.title}}</h1>
{{else}}
<h1>Create: {{page.title}}</h1>
{{endif}}

<form action="{{req.path_url}}" method="POST">
 <input type="hidden" name="mtime" value="{{page.mtime}}">
 Title: <input type="text" name="title" style="width: 70%" value="{{page.title}}"><br>
 Content: <input type="submit" value="Save">
 <a href="{{req.path_url}}">Cancel</a>
   <br>
 <textarea name="content" style="width: 100%; height: 75%" rows="40">{{page.content}}</textarea>
   <br>
 <input type="submit" value="Save">
 <a href="{{req.path_url}}">Cancel</a>
</form>
</body></html>
""")

class WikiApp(object):
    ...
```

```
    edit_template = EDIT_TEMPLATE

    def action_edit_GET(self, req, page):
        text = self.edit_template.substitute(
            page=page, req=req)
        return Response(text)
```

As you can see, all the action here is in the template.

In `<form action="{{req.path_url}}" method="POST">` we submit to `req.path_url`; that's everything *but* `?action=edit`. So we are POSTing right over the view page. This has the nice side effect of automatically invalidating any caches of the original page. It also is vaguely RESTful.

We save the last modified time in a hidden `mtime` field. This way we can detect concurrent updates. If start editing the page who's mtime is 100000, and someone else edits and saves a revision changing the mtime to 100010, we can use this hidden field to detect that conflict. Actually resolving the conflict is a little tricky and outside the scope of this particular tutorial, we'll just note the conflict to the user in an error.

From there we just have a very straight-forward HTML form. Note that we don't quote the values because that is done automatically by `HTMLTemplate`; if you are using something like `string.Template` or a templating language that doesn't do automatic quoting, you have to be careful to quote all the field values.

We don't have any error conditions in our application, but if there were error conditions we might have to re-display this form with the input values the user already gave. In that case we'd do something like:

```
<input type="text" name="title"
 value="{{req.params.get('title', page.title)}}">
```

This way we use the value in the request (`req.params` is both the query string parameters and any variables in a POST response), but if there is no value (e.g., first request) then we use the page values.

9.2.9 Processing the Form

The form submits to `action_view_POST` (`view` is the default action). So we have to implement that method:

```
class WikiApp(object):
    ...

    def action_view_POST(self, req, page):
        submit_mtime = int(req.params.get('mtime') or '0') or None
        if page.mtime != submit_mtime:
            return exc.HTTPPreconditionFailed(
                "The page has been updated since you started editing it")
        page.set(
            title=req.params['title'],
            content=req.params['content'])
        resp = exc.HTTPSeeOther(
            location=req.path_url)
        return resp
```

The first thing we do is check the mtime value. It can be an empty string (when there's no mtime, like when you are creating a page) or an integer. `int(req.params.get('time') or '0') or None` basically makes sure we don't pass `""` to `int()` (which is an error) then turns 0 into None (`0 or None` will evaluate to None in Python – `false_value or other_value` in Python resolves to `other_value`). If it fails we just give a not-very-helpful error message, using `412 Precondition Failed` (typically preconditions are HTTP headers like `If-Unmodified-Since`, but we can't really get the browser to send requests like that, so we use the hidden field instead).

Note: Error statuses in HTTP are often under-used because people think they need to either return an error (useful for machines) or an error message or interface (useful for humans). In fact you can do both: you can give any human readable error message with your error response.

One problem is that Internet Explorer will replace error messages with its own incredibly unhelpful error messages. However, it will only do this if the error message is short. If it's fairly large (4Kb is large enough) it will show the error message it was given. You can load your error with a big HTML comment to accomplish this, like `"<!-- %s -->" % ('x'*4000)`.

You can change the status of any response with `resp.status_int = 412`, or you can change the body of an `exc.HTTPSomething` with `resp.body = new_body`. The primary advantage of using the classes in `webob.exc` is giving the response a clear name and a boilerplate error message.

After we check the mtime we get the form parameters from `req.params` and issue a redirect back to the original view page. `303 See Other` is a good response to give after accepting a POST form submission, as it gets rid of the POST (no warning messages for the user if they try to go back).

In this example we've used `req.params` for all the form values. If we wanted to be specific about where we get the values from, they could come from `req.GET` (the query string, a misnomer since the query string is present even in POST requests) or `req.POST` (a POST form body). While sometimes it's nice to distinguish between these two locations, for the most part it doesn't matter. If you want to check the request method (e.g., make sure you can't change a page with a GET request) there's no reason to do it by accessing these method-specific getters. It's better to just handle the method specifically. We do it here by including the request method in our dispatcher (dispatching to `action_view_GET` or `action_view_POST`).

9.2.10 Cookies

One last little improvement we can do is show the user a message when they update the page, so it's not quite so mysteriously just another page view.

A simple way to do this is to set a cookie after the save, then display it in the page view. To set it on save, we add a little to `action_view_POST`:

```
def action_view_POST(self, req, page):
    ...
    resp = exc.HTTPSeeOther(
        location=req.path_url)
    resp.set_cookie('message', 'Page updated')
    return resp
```

And then in `action_view_GET`:

```
VIEW_TEMPLATE = HTMLTemplate("""\
...
{{if message}}
<div style="background-color: #99f">{{message}}</div>
{{endif}}
...""")

class WikiApp(object):
    ...

    def action_view_GET(self, req, page):
        ...
        if req.cookies.get('message'):
            message = req.cookies['message']
```

```
        else:
            message = None
        text = self.view_template.substitute(
            page=page, req=req, message=message)
        resp = Response(text)
        if message:
            resp.delete_cookie('message')
        else:
            resp.last_modified = page.mtime
            resp.conditional_response = True
        return resp
```

`req.cookies` is just a dictionary, and we also delete the cookie if it is present (so the message doesn't keep getting set). The conditional response stuff only applies when there isn't any message, as messages are private. Another alternative would be to display the message with Javascript, like:

```
<script type="text/javascript">
function readCookie(name) {
    var nameEQ = name + "=";
    var ca = document.cookie.split(';');
    for (var i=0; i < ca.length; i++) {
        var c = ca[i];
        while (c.charAt(0) == ' ') c = c.substring(1,c.length);
        if (c.indexOf(nameEQ) == 0) return c.substring(nameEQ.length,c.length);
    }
    return null;
}

function createCookie(name, value, days) {
    if (days) {
        var date = new Date();
        date.setTime(date.getTime()+(days*24*60*60*1000));
        var expires = "; expires="+date.toGMTString();
    } else {
        var expires = "";
    }
    document.cookie = name+"="+value+expires+"; path=/";
}

function eraseCookie(name) {
    createCookie(name, "", -1);
}

function showMessage() {
    var message = readCookie('message');
    if (message) {
        var el = document.getElementById('message');
        el.innerHTML = message;
        el.style.display = '';
        eraseCookie('message');
    }
}
</script>
```

Then put `<div id="messaage" style="display: none"></div>` in the page somewhere. This has the advantage of being very cacheable and simple on the server side.

9.2.11 Conclusion

We're done, hurrah!

9.3 Comment Example

Contents

9.3.1 Introduction

This is an example of how to write WSGI middleware with WebOb. The specific example adds a simple comment form to HTML web pages; any page served through the middleware that is HTML gets a comment form added to it, and shows any existing comments.

9.3.2 Code

The finished code for this is available in docs/comment-example-code/example.py – you can run that file as a script to try it out.

9.3.3 Instantiating Middleware

Middleware of any complexity at all is usually best created as a class with its configuration as arguments to that class.

Every middleware needs an application (`app`) that it wraps. This middleware also needs a location to store the comments; we'll put them all in a single directory.

```
import os

class Commenter(object):
    def __init__(self, app, storage_dir):
        self.app = app
        self.storage_dir = storage_dir
        if not os.path.exists(storage_dir):
            os.makedirs(storage_dir)
```

When you use this middleware, you'll use it like:

```
app = ... make the application ...
app = Commenter(app, storage_dir='./comments')
```

For our application we'll use a simple static file server that is included with Paste (use `easy_install Paste` to install this). The setup is all at the bottom of `example.py`, and looks like this:

```
if __name__ == '__main__':
    import optparse
    parser = optparse.OptionParser(
        usage='%prog --port=PORT BASE_DIRECTORY'
        )
    parser.add_option(
        '-p', '--port',
        default='8080',
        dest='port',
        type='int',
        help='Port to serve on (default 8080)')
    parser.add_option(
        '--comment-data',
        default='./comments',
        dest='comment_data',
        help='Place to put comment data into (default ./comments/)')
    options, args = parser.parse_args()
    if not args:
        parser.error('You must give a BASE_DIRECTORY')
    base_dir = args[0]
    from paste.urlparser import StaticURLParser
    app = StaticURLParser(base_dir)
    app = Commenter(app, options.comment_data)
    from wsgiref.simple_server import make_server
    httpd = make_server('localhost', options.port, app)
    print 'Serving on http://localhost:%s' % options.port
    try:
        httpd.serve_forever()
    except KeyboardInterrupt:
        print '^C'
```

I won't explain it here, but basically it takes some options, creates an application that serves static files (`StaticURLParser(base_dir)`), wraps it with `Commenter(app, options.comment_data)` then serves that.

9.3.4 The Middleware

While we've created the class structure for the middleware, it doesn't actually do anything. Here's a kind of minimal version of the middleware (using WebOb):

```
from webob import Request

class Commenter(object):

    def __init__(self, app, storage_dir):
        self.app = app
        self.storage_dir = storage_dir
        if not os.path.exists(storage_dir):
            os.makedirs(storage_dir)

    def __call__(self, environ, start_response):
        req = Request(environ)
        resp = req.get_response(self.app)
        return resp(environ, start_response)
```

This doesn't modify the response it any way. You could write it like this without WebOb:

```
class Commenter(object):
    ...
    def __call__(self, environ, start_response):
        return self.app(environ, start_response)
```

But it won't be as convenient later. First, lets create a little bit of infrastructure for our middleware. We need to save and load per-url data (the comments themselves). We'll keep them in pickles, where each url has a pickle named after the url (but double-quoted, so `http://localhost:8080/index.html` becomes `http%3A%2F%2Flocalhost%3A8080%2Findex.html`).

```
from cPickle import load, dump

class Commenter(object):
    ...

    def get_data(self, url):
        filename = self.url_filename(url)
        if not os.path.exists(filename):
            return []
        else:
            f = open(filename, 'rb')
            data = load(f)
            f.close()
            return data

    def save_data(self, url, data):
        filename = self.url_filename(url)
        f = open(filename, 'wb')
        dump(data, f)
        f.close()

    def url_filename(self, url):
        # Double-quoting makes the filename safe
        return os.path.join(self.storage_dir, urllib.quote(url, ''))
```

You can get the full request URL with `req.url`, so to get the comment data with these methods you do `data = self.get_data(req.url)`.

Now we'll update the `__call__` method to filter *some* responses, and get the comment data for those. We don't want to change responses that were error responses (anything but `200`), nor do we want to filter responses that aren't HTML. So we get:

```
class Commenter(object):
    ...

    def __call__(self, environ, start_response):
        req = Request(environ)
        resp = req.get_response(self.app)
        if resp.content_type != 'text/html' or resp.status_code != 200:
            return resp(environ, start_response)
        data = self.get_data(req.url)
        ... do stuff with data, update resp ...
        return resp(environ, start_response)
```

So far we're punting on actually adding the comments to the page. We also haven't defined what `data` will hold. Let's say it's a list of dictionaries, where each dictionary looks like `{'name': 'John Doe', 'homepage': 'http://blog.johndoe.com', 'comments': 'Great site!'}`.

We'll also need a simple method to add stuff to the page. We'll use a regular expression to find the end of the page and put text in:

```
import re

class Commenter(object):
    ...

    _end_body_re = re.compile(r'</body.*?>', re.I|re.S)

    def add_to_end(self, html, extra_html):
        """
        Adds extra_html to the end of the html page (before </body>)
        """
        match = self._end_body_re.search(html)
        if not match:
            return html + extra_html
        else:
            return html[:match.start()] + extra_html + html[match.start():]
```

And then we'll use it like:

```
data = self.get_data(req.url)
body = resp.body
body = self.add_to_end(body, self.format_comments(data))
resp.body = body
return resp(environ, start_response)
```

We get the body, update it, and put it back in the response. This also updates `Content-Length`. Then we define:

```
from webob import html_escape

class Commenter(object):
    ...

    def format_comments(self, comments):
        if not comments:
            return ''
        text = []
        text.append('<hr>')
        text.append('<h2><a name="comment-area"></a>Comments (%s):</h2>' % len(comments))
        for comment in comments:
            text.append('<h3><a href="%s">%s</a> at %s:</h3>' % (
                html_escape(comment['homepage']), html_escape(comment['name']),
                time.strftime('%c', comment['time'])))
            # Susceptible to XSS attacks!:
            text.append(comment['comments'])
        return ''.join(text)
```

We put in a header (with an anchor we'll use later), and a section for each comment. Note that `html_escape` is the same as `cgi.escape` and just turns `&` into `&`, etc.

Because we put in some text without quoting it is susceptible to a Cross-Site Scripting attack. Fixing that is beyond the scope of this tutorial; you could quote it or clean it with something like lxml.html.clean.

9.3.5 Accepting Comments

All of those pieces *display* comments, but still no one can actually make comments. To handle this we'll take a little piece of the URL space for our own, everything under `/.comments`, so when someone POSTs there it will add a

comment.

When the request comes in there are two parts to the path: `SCRIPT_NAME` and `PATH_INFO`. Everything in `SCRIPT_NAME` has already been parsed, and everything in `PATH_INFO` has yet to be parsed. That means that the URL *without* `PATH_INFO` is the path to the middleware; we can intercept anything else below `SCRIPT_NAME` but nothing above it. The name for the URL without `PATH_INFO` is `req.application_url`. We have to capture it early to make sure it doesn't change (since the WSGI application we are wrapping may update `SCRIPT_NAME` and `PATH_INFO`).

So here's what this all looks like:

```
class Commenter(object):
    ...

    def __call__(self, environ, start_response):
        req = Request(environ)
        if req.path_info_peek() == '.comments':
            return self.process_comment(req)(environ, start_response)
        # This is the base path of *this* middleware:
        base_url = req.application_url
        resp = req.get_response(self.app)
        if resp.content_type != 'text/html' or resp.status_code != 200:
            # Not an HTML response, we don't want to
            # do anything to it
            return resp(environ, start_response)
        # Make sure the content isn't gzipped:
        resp.decode_content()
        comments = self.get_data(req.url)
        body = resp.body
        body = self.add_to_end(body, self.format_comments(comments))
        body = self.add_to_end(body, self.submit_form(base_url, req))
        resp.body = body
        return resp(environ, start_response)
```

`base_url` is the path where the middleware is located (if you run the example server, it will be `http://localhost:PORT/`). We use `req.path_info_peek()` to look at the next segment of the URL – what comes after base_url. If it is `.comments` then we handle it internally and don't pass the request on.

We also put in a little guard, `resp.decode_content()` in case the application returns a gzipped response.

Then we get the data, add the comments, add the *form* to make new comments, and return the result.

submit_form

Here's what the form looks like:

```
class Commenter(object):
    ...

    def submit_form(self, base_path, req):
        return '''<h2>Leave a comment:</h2>
        <form action="%s/.comments" method="POST">
         <input type="hidden" name="url" value="%s">
         <table width="100%%">
          <tr><td>Name:</td>
              <td><input type="text" name="name" style="width: 100%%"></td></tr>
          <tr><td>URL:</td>
              <td><input type="text" name="homepage" style="width: 100%%"></td></tr>
         </table>
```

```
        Comments:<br>
        <textarea name="comments" rows=10 style="width: 100%%"></textarea><br>
        <input type="submit" value="Submit comment">
      </form>
      ''' % (base_path, html_escape(req.url))
```

Nothing too exciting. It submits a form with the keys `url` (the URL being commented on), `name`, `homepage`, and `comments`.

process_comment

If you look at the method call, what we do is call the method then treat the result as a WSGI application:

```
return self.process_comment(req)(environ, start_response)
```

You could write this as:

```
response = self.process_comment(req)
return response(environ, start_response)
```

A common pattern in WSGI middleware that *doesn't* use WebOb is to just do:

```
return self.process_comment(environ, start_response)
```

But the WebOb style makes it easier to modify the response if you want to; modifying a traditional WSGI response/application output requires changing your logic flow considerably.

Here's the actual processing code:

```
from webob import exc
from webob import Response

class Commenter(object):
    ...

    def process_comment(self, req):
        try:
            url = req.params['url']
            name = req.params['name']
            homepage = req.params['homepage']
            comments = req.params['comments']
        except KeyError, e:
            resp = exc.HTTPBadRequest('Missing parameter: %s' % e)
            return resp
        data = self.get_data(url)
        data.append(dict(
            name=name,
            homepage=homepage,
            comments=comments,
            time=time.gmtime()))
        self.save_data(url, data)
        resp = exc.HTTPSeeOther(location=url+'#comment-area')
        return resp
```

We either give a Bad Request response (if the form submission is somehow malformed), or a redirect back to the original page.

The classes in `webob.exc` (like `HTTPBadRequest` and `HTTPSeeOther`) are Response subclasses that can be used to quickly create responses for these non-200 cases where the response body usually doesn't matter much.

9.3.6 Conclusion

This shows how to make response modifying middleware, which is probably the most difficult kind of middleware to write with WSGI – modifying the request is quite simple in comparison, as you simply update `environ`.

9.4 JSON-RPC Example

Contents

author Ian Bicking

9.4.1 Introduction

This is an example of how to write a web service using WebOb. The example shows how to create a JSON-RPC endpoint using WebOb and the simplejson JSON library. This also shows how to use WebOb as a client library using WSGIProxy.

While this example presents JSON-RPC, this is not an endorsement of JSON-RPC. In fact I don't like JSON-RPC. It's unnecessarily un-RESTful, and modelled too closely on XML-RPC.

9.4.2 Code

The finished code for this is available in docs/json-example-code/jsonrpc.py – you can run that file as a script to try it out, or import it.

9.4.3 Concepts

JSON-RPC wraps an object, allowing you to call methods on that object and get the return values. It also provides a way to get error responses.

The specification goes into the details (though in a vague sort of way). Here's the basics:

- All access goes through a POST to a single URL.
- The POST contains a JSON body that looks like:

```
    {"method": "methodName",
     "id": "arbitrary-something",
     "params": [arg1, arg2, ...]}
```

- The `id` parameter is just a convenience for the client to keep track of which response goes with which request. This makes asynchronous calls (like an XMLHttpRequest) easier. We just send the exact same id back as we get, we never look at it.
- The response is JSON. A successful response looks like:

```
    {"result": the_result,
     "error": null,
     "id": "arbitrary-something"}
```

- The error response looks like:

```
    {"result": null,
     "error": {"name": "JSONRPCError",
               "code": (number 100-999),
               "message": "Some Error Occurred",
               "error": "whatever you want\n(a traceback?)"},
     "id": "arbitrary-something"}
```

- It doesn't seem to indicate if an error response should have a 200 response or a 500 response. So as not to be completely stupid about HTTP, we choose a 500 resonse, as giving an error with a 200 response is irresponsible.

9.4.4 Infrastructure

To make this easier to test, we'll set up a bit of infrastructure. This will open up a server (using wsgiref) and serve up our application (note that *creating* the application is left out to start with):

```
import sys

def main(args=None):
    import optparse
    from wsgiref import simple_server
    parser = optparse.OptionParser(
        usage="%prog [OPTIONS] MODULE:EXPRESSION")
    parser.add_option(
        '-p', '--port', default='8080',
        help='Port to serve on (default 8080)')
    parser.add_option(
        '-H', '--host', default='127.0.0.1',
        help='Host to serve on (default localhost; 0.0.0.0 to make public)')
    if args is None:
        args = sys.argv[1:]
    options, args = parser.parse_args()
    if not args or len(args) > 1:
        print 'You must give a single object reference'
        parser.print_help()
        sys.exit(2)
    app = make_app(args[0])
    server = simple_server.make_server(
        options.host, int(options.port),
        app)
    print 'Serving on http://%s:%s' % (options.host, options.port)
    server.serve_forever()
```

```
if __name__ == '__main__':
    main()
```

I won't describe this much. It starts a server, serving up just the app created by `make_app(args[0])`. `make_app` will have to load up the object and wrap it in our WSGI/WebOb wrapper. We'll be calling that wrapper `JSONRPC(obj)`, so here's how it'll go:

```
def make_app(expr):
    module, expression = expr.split(':', 1)
    __import__(module)
    module = sys.modules[module]
    obj = eval(expression, module.__dict__)
    return JsonRpcApp(obj)
```

We use `__import__(module)` to import the module, but its return value is wonky. We can find the thing it imported in `sys.modules` (a dictionary of all the loaded modules). Then we evaluate the second part of the expression in the namespace of the module. This lets you do something like `smtplib:SMTP('localhost')` to get a fully instantiated SMTP object.

That's all the infrastructure we'll need for the server side. Now we just have to implement `JsonRpcApp`.

9.4.5 The Application Wrapper

Note that I'm calling this an "application" because that's the terminology WSGI uses. Everything that gets *called* is an "application", and anything that calls an application is called a "server".

The instantiation of the server is already figured out:

```
class JsonRpcApp(object):

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

    def __call__(self, environ, start_response):
        ... the WSGI interface ...
```

So the server is an instance bound to the particular object being exposed, and `__call__` implements the WSGI interface.

We'll start with a simple outline of the WSGI interface, using a kind of standard WebOb setup:

```
from webob import Request, Response
from webob import exc

class JsonRpcApp(object):
    ...
    def __call__(self, environ, start_response):
        req = Request(environ)
        try:
            resp = self.process(req)
        except ValueError, e:
            resp = exc.HTTPBadRequest(str(e))
        except exc.HTTPException, e:
            resp = e
        return resp(environ, start_response)
```

We first create a request object. The request object just wraps the WSGI environment. Then we create the response object in the `process` method (which we still have to write). We also do some exception catching. We'll turn any `ValueError` into a `400 Bad Request` response. We'll also let `process` raise any

`web.exc.HTTPException` exception. There's an exception defined in that module for all the HTTP error responses, like `405 Method Not Allowed`. These exceptions are themselves WSGI applications (as is `webob.Response`), and so we call them like WSGI applications and return the result.

9.4.6 The `process` method

The `process` method of course is where all the fancy stuff happens. We'll start with just the most minimal implementation, with no error checking or handling:

```
from simplejson import loads, dumps

class JsonRpcApp(object):
    ...
    def process(self, req):
        json = loads(req.body)
        method = json['method']
        params = json['params']
        id = json['id']
        method = getattr(self.obj, method)
        result = method(*params)
        resp = Response(
            content_type='application/json',
            body=dumps(dict(result=result,
                            error=None,
                            id=id)))
        return resp
```

As long as the request is properly formed and the method doesn't raise any exceptions, you are pretty much set. But of course that's not a reasonable expectation. There's a whole bunch of things that can go wrong. For instance, it has to be a POST method:

```
if not req.method == 'POST':
    raise exc.HTTPMethodNotAllowed(
        "Only POST allowed",
        allowed='POST')
```

And maybe the request body doesn't contain valid JSON:

```
try:
    json = loads(req.body)
except ValueError, e:
    raise ValueError('Bad JSON: %s' % e)
```

And maybe all the keys aren't in the dictionary:

```
try:
    method = json['method']
    params = json['params']
    id = json['id']
except KeyError, e:
    raise ValueError(
        "JSON body missing parameter: %s" % e)
```

And maybe it's trying to acces a private method (a method that starts with _) – that's not just a bad request, we'll call that case `403 Forbidden`.

```
if method.startswith('_'):
    raise exc.HTTPForbidden(
        "Bad method name %s: must not start with _" % method)
```

```
```

And maybe `json['params']` isn't a list:

```
if not isinstance(params, list):
    raise ValueError(
        "Bad params %r: must be a list" % params)
```

And maybe the method doesn't exist:

```
try:
    method = getattr(self.obj, method)
except AttributeError:
    raise ValueError(
        "No such method %s" % method)
```

The last case is the error we actually can expect: that the method raises some exception.

```
try:
    result = method(*params)
except:
    tb = traceback.format_exc()
    exc_value = sys.exc_info()[1]
    error_value = dict(
        name='JSONRPCError',
        code=100,
        message=str(exc_value),
        error=tb)
    return Response(
        status=500,
        content_type='application/json',
        body=dumps(dict(result=None,
                        error=error_value,
                        id=id)))
```

That's a complete server.

9.4.7 The Complete Code

Since we showed all the error handling in pieces, here's the complete code:

```
from webob import Request, Response
from webob import exc
from simplejson import loads, dumps
import traceback
import sys

class JsonRpcApp(object):
    """
    Serve the given object via json-rpc (http://json-rpc.org/)
    """

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

    def __call__(self, environ, start_response):
        req = Request(environ)
        try:
            resp = self.process(req)
```

```
        except ValueError, e:
            resp = exc.HTTPBadRequest(str(e))
        except exc.HTTPException, e:
            resp = e
        return resp(environ, start_response)

    def process(self, req):
        if not req.method == 'POST':
            raise exc.HTTPMethodNotAllowed(
                "Only POST allowed",
                allowed='POST')
        try:
            json = loads(req.body)
        except ValueError, e:
            raise ValueError('Bad JSON: %s' % e)
        try:
            method = json['method']
            params = json['params']
            id = json['id']
        except KeyError, e:
            raise ValueError(
                "JSON body missing parameter: %s" % e)
        if method.startswith('_'):
            raise exc.HTTPForbidden(
                "Bad method name %s: must not start with _" % method)
        if not isinstance(params, list):
            raise ValueError(
                "Bad params %r: must be a list" % params)
        try:
            method = getattr(self.obj, method)
        except AttributeError:
            raise ValueError(
                "No such method %s" % method)
        try:
            result = method(*params)
        except:
            text = traceback.format_exc()
            exc_value = sys.exc_info()[1]
            error_value = dict(
                name='JSONRPCError',
                code=100,
                message=str(exc_value),
                error=text)
            return Response(
                status=500,
                content_type='application/json',
                body=dumps(dict(result=None,
                                error=error_value,
                                id=id)))
        return Response(
            content_type='application/json',
            body=dumps(dict(result=result,
                            error=None,
                            id=id)))
```

9.4.8 The Client

It would be nice to have a client to test out our server. Using WSGIProxy we can use WebOb Request and Response to do actual HTTP connections.

The basic idea is that you can create a blank Request:

```
>>> from webob import Request
>>> req = Request.blank('http://python.org')
```

Then you can send that request to an application:

```
>>> from wsgiproxy.exactproxy import proxy_exact_request
>>> resp = req.get_response(proxy_exact_request)
```

This particular application (`proxy_exact_request`) sends the request over HTTP:

```
>>> resp.content_type
'text/html'
>>> resp.body[:10]
'<!DOCTYPE '
```

So we're going to create a proxy object that constructs WebOb-based jsonrpc requests, and sends those using `proxy_exact_request`.

9.4.9 The Proxy Client

The proxy client is instantiated with its base URL. We'll also let you pass in a proxy application, in case you want to do local requests (e.g., to do direct tests against a `JsonRpcApp` instance):

```
class ServerProxy(object):

    def __init__(self, url, proxy=None):
        self._url = url
        if proxy is None:
            from wsgiproxy.exactproxy import proxy_exact_request
            proxy = proxy_exact_request
        self.proxy = proxy
```

This ServerProxy object itself doesn't do much, but you can call methods on it. We can intercept any access `ServerProxy(...).method` with the magic function `__getattr__`. Whenever you get an attribute that doesn't exist in an instance, Python will call `inst.__getattr__(attr_name)` and return that. When you *call* a method, you are calling the object that `.method` returns. So we'll create a helper object that is callable, and our `__getattr__` will just return that:

```
class ServerProxy(object):
    ...
    def __getattr__(self, name):
        # Note, even attributes like __contains__ can get routed
        # through __getattr__
        if name.startswith('_'):
            raise AttributeError(name)
        return _Method(self, name)

class _Method(object):
    def __init__(self, parent, name):
        self.parent = parent
        self.name = name
```

Now when we call the method we'll be calling `_Method.__call__`, and the HTTP endpoint will be `self.parent._url`, and the method name will be `self.name`.

Here's the code to do the call:

```
class _Method(object):
    ...

    def __call__(self, *args):
        json = dict(method=self.name,
                    id=None,
                    params=list(args))
        req = Request.blank(self.parent._url)
        req.method = 'POST'
        req.content_type = 'application/json'
        req.body = dumps(json)
        resp = req.get_response(self.parent.proxy)
        if resp.status_code != 200 and not (
            resp.status_code == 500
            and resp.content_type == 'application/json'):
            raise ProxyError(
                "Error from JSON-RPC client %s: %s"
                % (self._url, resp.status),
                resp)
        json = loads(resp.body)
        if json.get('error') is not None:
            e = Fault(
                json['error'].get('message'),
                json['error'].get('code'),
                json['error'].get('error'),
                resp)
            raise e
        return json['result']
```

We raise two kinds of exceptions here. `ProxyError` is when something unexpected happens, like a `404 Not Found`. `Fault` is when a more expected exception occurs, i.e., the underlying method raised an exception.

In both cases we'll keep the response object around, as that can be interesting. Note that you can make exceptions have any methods or signature you want, which we'll do:

```
class ProxyError(Exception):
    """
    Raised when a request via ServerProxy breaks
    """
    def __init__(self, message, response):
        Exception.__init__(self, message)
        self.response = response

class Fault(Exception):
    """
    Raised when there is a remote error
    """
    def __init__(self, message, code, error, response):
        Exception.__init__(self, message)
        self.code = code
        self.error = error
        self.response = response
    def __str__(self):
        return 'Method error calling %s: %s\n%s' % (
            self.response.request.url,
```

```
            self.args[0],
            self.error)
```

9.4.10 Using Them Together

Good programmers start with tests. But at least we'll end with a test. We'll use doctest for our tests. The test is in docs/json-example-code/test_jsonrpc.txt and you can run it with docs/json-example-code/test_jsonrpc.py, which looks like:

```
if __name__ == '__main__':
    import doctest
    doctest.testfile('test_jsonrpc.txt')
```

As you can see, it's just a stub to run the doctest. We'll need a simple object to expose. We'll make it real simple:

```
>>> class Divider(object):
...     def divide(self, a, b):
...         return a / b
```

Then we'll get the app setup:

```
>>> from jsonrpc import *
>>> app = JsonRpcApp(Divider())
```

And attach the client *directly* to it:

```
>>> proxy = ServerProxy('http://localhost:8080', proxy=app)
```

Because we gave the app itself as the proxy, the URL doesn't actually matter.

Now, if you are used to testing you might ask: is this kosher? That is, we are shortcircuiting HTTP entirely. Is this a realistic test?

One thing you might be worried about in this case is that there are more shared objects than you'd have with HTTP. That is, everything over HTTP is serialized to headers and bodies. Without HTTP, we can send stuff around that can't go over HTTP. This *could* happen, but we're mostly protected because the only thing the application's share is the WSGI `environ`. Even though we use a `webob.Request` object on both side, it's not the *same* request object, and all the state is studiously kept in the environment. We *could* share things in the environment that couldn't go over HTTP. For instance, we could set `environ['jsonrpc.request_value'] = dict(...)`, and avoid `simplejson.dumps` and `simplejson.loads`. We *could* do that, and if we did then it is possible our test would work even though the libraries were broken over HTTP. But of course inspection shows we *don't* do that. A little discipline is required to resist playing clever tricks (or else you can play those tricks and do more testing). Generally it works well.

So, now we have a proxy, lets use it:

```
>>> proxy.divide(10, 4)
2
>>> proxy.divide(10, 4.0)
2.5
```

Lastly, we'll test a couple error conditions. First a method error:

```
>>> proxy.divide(10, 0)
Traceback (most recent call last):
  ...
Fault: Method error calling http://localhost:8080: integer division or modulo by zero
Traceback (most recent call last):
```

```
  File ...
    result = method(*params)
  File ...
    return a / b
ZeroDivisionError: integer division or modulo by zero
```

It's hard to actually predict this exception, because the test of the exception itself contains the traceback from the underlying call, with filenames and line numbers that aren't stable. We use `# doctest: +ELLIPSIS` so that we can replace text we don't care about with `...`. This is actually figured out through copy-and-paste, and visual inspection to make sure it looks sensible.

The other exception can be:

```
>>> proxy.add(1, 1)
Traceback (most recent call last):
    ...
ProxyError: Error from JSON-RPC client http://localhost:8080: 400 Bad Request
```

Here the exception isn't a JSON-RPC method exception, but a more basic ProxyError exception.

9.4.11 Conclusion

Hopefully this will give you ideas about how to implement web services of different kinds using WebOb. I hope you also can appreciate the elegance of the symmetry of the request and response objects, and the client and server for the protocol.

Many of these techniques would be better used with a RESTful service, so do think about that direction if you are implementing your own protocol.

9.5 Another Do-It-Yourself Framework

Contents

9.5.1 Introduction and Audience

It's been over two years since I wrote the first version of this tutorial. I decided to give it another run with some of the tools that have come about since then (particularly WebOb).

Sometimes Python is accused of having too many web frameworks. And it's true, there are a lot. That said, I think writing a framework is a useful exercise. It doesn't let you skip over too much without understanding it. It removes the magic. So even if you go on to use another existing framework (which I'd probably advise you do), you'll be able to understand it better if you've written something like it on your own.

This tutorial shows you how to create a web framework of your own, using WSGI and WebOb. No other libraries will be used.

For the longer sections I will try to explain any tricky parts on a line-by line basis following the example.

9.5.2 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. That's the simplest summary, but there is more – WSGI lets you add annotation to the request, and adds some more metadata to the request.

WSGI more specifically is made up of an *application* and a *server*. The application is a function that receives the request and produces the response. The server is the thing that calls the application function.

A very simple application looks like this:

```
>>> def application(environ, start_response):
...     start_response('200 OK', [('Content-Type', 'text/html')])
...     return ['Hello World!']
```

The `environ` argument is a dictionary with values like the environment in a CGI request. The header `Host:`, for instance, goes in `environ['HTTP_HOST']`. The path is in `environ['SCRIPT_NAME']` (which is the path leading *up to* the application), and `environ['PATH_INFO']` (the remaining path that the application should interpret).

We won't focus much on the server, but we will use WebOb to handle the application. WebOb in a way has a simple server interface. To use it you create a new request with `req = webob.Request.blank('http://localhost/test')`, and then call the application with `resp = req.get_response(app)`. For example:

```
>>> from webob import Request
>>> req = Request.blank('http://localhost/test')
>>> resp = req.get_response(application)
>>> print resp
200 OK
Content-Type: text/html

Hello World!
```

This is an easy way to test applications, and we'll use it to test the framework we're creating.

9.5.3 About WebOb

WebOb is a library to create a request and response object. It's centered around the WSGI model. Requests are wrappers around the environment. For example:

```
>>> req = Request.blank('http://localhost/test')
>>> req.environ['HTTP_HOST']
'localhost:80'
>>> req.host
'localhost:80'
>>> req.path_info
'/test'
```

Responses are objects that represent the... well, response. The status, headers, and body:

```
>>> from webob import Response
>>> resp = Response(body='Hello World!')
>>> resp.content_type
'text/html'
>>> resp.content_type = 'text/plain'
>>> print resp
200 OK
Content-Length: 12
Content-Type: text/plain; charset=UTF-8

Hello World!
```

Responses also happen to be WSGI applications. That means you can call `resp(environ, start_response)`. Of course it's much less *dynamic* than a normal WSGI application.

These two pieces solve a lot of the more tedious parts of making a framework. They deal with parsing most HTTP headers, generating valid responses, and a number of unicode issues.

9.5.4 Serving Your Application

While we can test the application using WebOb, you might want to serve the application. Here's the basic recipe, using the Paste HTTP server:

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

you could also use wsgiref from the standard library, but this is mostly appropriate for testing as it is single-threaded:

```
if __name__ == '__main__':
    from wsgiref.simple_server import make_server
    server = make_server('127.0.0.1', 8080, app)
    server.serve_forever()
```

9.5.5 Making A Framework

Well, now we need to start work on our framework.

Here's the basic model we'll be creating:

- We'll define routes that point to controllers
- We'll create a simple framework for creating controllers

9.5.6 Routing

We'll use explicit routes using URI templates (minus the domains) to match paths. We'll add a little extension that you can use `{name:regular expression}`, where the named segment must then match that regular expression. The matches will include a "controller" variable, which will be a string like "module_name:function_name". For our examples we'll use a simple blog.

So here's what a route would look like:

```
app = Router()
app.add_route('/', controller='controllers:index')
app.add_route('/{year:\d\d\d\d}/',
              controller='controllers:archive')
app.add_route('/{year:\d\d\d\d}/{month:\d\d}/',
              controller='controllers:archive')
app.add_route('/{year:\d\d\d\d}/{month:\d\d}/{slug}',
              controller='controllers:view')
app.add_route('/post', controller='controllers:post')
```

To do this we'll need a couple pieces:

- Something to match those URI template things.
- Something to load the controller
- The object to patch them together (`Router`)

Routing: Templates

To do the matching, we'll compile those templates to regular expressions.

```
1  >>> import re
2  >>> var_regex = re.compile(r'''
3  ...     \{          # The exact character "{"
4  ...     (\w+)       # The variable name (restricted to a-z, 0-9, _)
5  ...     (?::([^}]+))? # The optional :regex part
6  ...     \}          # The exact character "}"
7  ...     ''', re.VERBOSE)
8  >>> def template_to_regex(template):
9  ...     regex = ''
10 ...     last_pos = 0
11 ...     for match in var_regex.finditer(template):
12 ...         regex += re.escape(template[last_pos:match.start()])
13 ...         var_name = match.group(1)
14 ...         expr = match.group(2) or '[^/]+'
15 ...         expr = '(?P<%s>%s)' % (var_name, expr)
16 ...         regex += expr
17 ...         last_pos = match.end()
18 ...     regex += re.escape(template[last_pos:])
19 ...     regex = '^%s$' % regex
20 ...     return regex
```

line 2: Here we create the regular expression. The `re.VERBOSE` flag makes the regular expression parser ignore whitespace and allow comments, so we can avoid some of the feel of line-noise. This matches any variables, i.e., `{var:regex}` (where `:regex` is optional). Note that there are two groups we capture: `match.group(1)` will be the variable name, and `match.group(2)` will be the regular expression (or None when there is no regular expression). Note that `(?:...)?` means that the section is optional.

line 9: This variable will hold the regular expression that we are creating.

line 10: This contains the position of the end of the last match.

line 11: The `finditer` method yields all the matches.

line 12: We're getting all the non-`{}` text from after the last match, up to the beginning of this match. We call `re.escape` on that text, which escapes any characters that have special meaning. So `.html` will be escaped as `\.html`.

line 13: The first match is the variable name.

line 14: `expr` is the regular expression we'll match against, the optional second match. The default is `[^/]+`, which matches any non-empty, non-/ string. Which seems like a reasonable default to me.

line 15: Here we create the actual regular expression. `(?P<name>...)` is a grouped expression that is named. When you get a match, you can look at `match.groupdict()` and get the names and values.

line 16, 17: We add the expression on to the complete regular expression and save the last position.

line 18: We add remaining non-variable text to the regular expression.

line 19: And then we make the regular expression match the complete string (`^` to force it to match from the start, `$` to make sure it matches up to the end).

To test it we can try some translations. You could put these directly in the docstring of the `template_to_regex` function and use doctest to test that. But I'm using doctest to test *this* document, so I can't put a docstring doctest inside the doctest itself. Anyway, here's what a test looks like:

```
>>> print template_to_regex('/a/static/path')
^\/a\/static\/path$
>>> print template_to_regex('/{year:\d\d\d\d}/{month:\d\d}/{slug}')
^\/(?P<year>\d\d\d\d)\/(?P<month>\d\d)\/(?P<slug>[^/]+)$
```

Routing: controller loading

To load controllers we have to import the module, then get the function out of it. We'll use the `__import__` builtin to import the module. The return value of `__import__` isn't very useful, but it puts the module into `sys.modules`, a dictionary of all the loaded modules.

Also, some people don't know how exactly the string method `split` works. It takes two arguments – the first is the character to split on, and the second is the maximum number of splits to do. We want to split on just the first `:` character, so we'll use a maximum number of splits of 1.

```
>>> import sys
>>> def load_controller(string):
...     module_name, func_name = string.split(':', 1)
...     __import__(module_name)
...     module = sys.modules[module_name]
...     func = getattr(module, func_name)
...     return func
```

Routing: putting it together

Now, the `Router` class. The class has the `add_route` method, and also a `__call__` method. That `__call__` method makes the Router object itself a WSGI application. So when a request comes in, it looks at `PATH_INFO` (also known as `req.path_info`) and hands off the request to the controller that matches that path.

```
1  >>> from webob import Request
2  >>> from webob import exc
3  >>> class Router(object):
```

```
4  ...     def __init__(self):
5  ...         self.routes = []
6  ...
7  ...     def add_route(self, template, controller, **vars):
8  ...         if isinstance(controller, basestring):
9  ...             controller = load_controller(controller)
10 ...         self.routes.append((re.compile(template_to_regex(template)),
11 ...                             controller,
12 ...                             vars))
13 ...
14 ...     def __call__(self, environ, start_response):
15 ...         req = Request(environ)
16 ...         for regex, controller, vars in self.routes:
17 ...             match = regex.match(req.path_info)
18 ...             if match:
19 ...                 req.urlvars = match.groupdict()
20 ...                 req.urlvars.update(vars)
21 ...                 return controller(environ, start_response)
22 ...         return exc.HTTPNotFound()(environ, start_response)
```

line 5: We are going to keep the route options in an ordered list. Each item will be `(regex, controller, vars)`: `regex` is the regular expression object to match against, `controller` is the controller to run, and `vars` are any extra (constant) variables.

line 8, 9: We will allow you to call `add_route` with a string (that will be imported) or a controller object. We test for a string here, and then import it if necessary.

line 14: Here we add a `__call__` method. This is the method used when you call an object like a function. You should recognize this as the WSGI signature.

line 15: We create a request object. Note we'll only use this request object in this function; if the controller wants a request object it'll have to make on of its own.

line 17: We test the regular expression against `req.path_info`. This is the same as `environ['PATH_INFO']`. That's all the request path left to be processed.

line 19: We set `req.urlvars` to the dictionary of matches in the regular expression. This variable actually maps to `environ['wsgiorg.routing_args']`. Any attributes you set on a request will, in one way or another, map to the environment dictionary: the request holds no state of its own.

line 20: We also add in any explicit variables passed in through `add_route()`.

line 21: Then we call the controller as a WSGI application itself. Any fancy framework stuff the controller wants to do, it'll have to do itself.

line 22: If nothing matches, we return a 404 Not Found response. `webob.exc.HTTPNotFound()` is a WSGI application that returns 404 responses. You could add a message too, like `webob.exc.HTTPNotFound('No route matched')`. Then, of course, we call the application.

9.5.7 Controllers

The router just passes the request on to the controller, so the controllers are themselves just WSGI applications. But we'll want to set up something to make those applications friendlier to write.

To do that we'll write a decorator. A decorator is a function that wraps another function. After decoration the function will be a WSGI application, but it will be decorating a function with a signature like `controller_func(req, **urlvars)`. The controller function will return a response object (which, remember, is a WSGI application on its own).

```
>>> from webob import Request, Response
>>> from webob import exc
>>> def controller(func):
...     def replacement(environ, start_response):
...         req = Request(environ)
...         try:
...             resp = func(req, **req.urlvars)
...         except exc.HTTPException, e:
...             resp = e
...         if isinstance(resp, basestring):
...             resp = Response(body=resp)
...         return resp(environ, start_response)
...     return replacement
```

line 3: This is the typical signature for a decorator – it takes one function as an argument, and returns a wrapped function.

line 4: This is the replacement function we'll return. This is called a closure – this function will have access to `func`, and everytime you decorate a new function there will be a new `replacement` function with its own value of `func`. As you can see, this is a WSGI application.

line 5: We create a request.

line 6: Here we catch any `webob.exc.HTTPException` exceptions. This is so you can do `raise webob.exc.HTTPNotFound()` in your function. These exceptions are themselves WSGI applications.

line 7: We call the function with the request object, any any variables in `req.urlvars`. And we get back a response.

line 10: We'll allow the function to return a full response object, or just a string. If they return a string, we'll create a `Response` object with that (and with the standard `200 OK` status, `text/html` content type, and `utf8` charset/encoding).

line 12: We pass the request on to the response. Which *also* happens to be a WSGI application. WSGI applications are falling from the sky!

line 13: We return the function object itself, which will take the place of the function.

You use this controller like:

```
>>> @controller
... def index(req):
...     return 'This is the index'
```

9.5.8 Putting It Together

Now we'll show a basic application. Just a hello world application for now. Note that this document is the module `__main__`.

```
>>> @controller
... def hello(req):
...     if req.method == 'POST':
...         return 'Hello %s!' % req.params['name']
...     elif req.method == 'GET':
...         return '''<form method="POST">
...             You're name: <input type="text" name="name">
...             <input type="submit">
...             </form>'''
>>> hello_world = Router()
>>> hello_world.add_route('/', controller=hello)
```

Now let's test that application:

```
>>> req = Request.blank('/')
>>> resp = req.get_response(hello_world)
>>> print resp
200 OK
Content-Type: text/html; charset=UTF-8
Content-Length: 131

<form method="POST">
            You're name: <input type="text" name="name">
            <input type="submit">
            </form>
>>> req.method = 'POST'
>>> req.body = 'name=Ian'
>>> resp = req.get_response(hello_world)
>>> print resp
200 OK
Content-Type: text/html; charset=UTF-8
Content-Length: 10

Hello Ian!
```

9.5.9 Another Controller

There's another pattern that might be interesting to try for a controller. Instead of a function, we can make a class with methods like `get`, `post`, etc. The `urlvars` will be used to instantiate the class.

We could do this as a superclass, but the implementation will be more elegant as a wrapper, like the decorator is a wrapper. Python 3.0 will add class decorators which will work like this.

We'll allow an extra `action` variable, which will define the method (actually `action_method`, where `_method` is the request method). If no action is given, we'll use just the method (i.e., `get`, `post`, etc).

```
 1  >>> def rest_controller(cls):
 2  ...     def replacement(environ, start_response):
 3  ...         req = Request(environ)
 4  ...         try:
 5  ...             instance = cls(req, **req.urlvars)
 6  ...             action = req.urlvars.get('action')
 7  ...             if action:
 8  ...                 action += '_' + req.method.lower()
 9  ...             else:
10  ...                 action = req.method.lower()
11  ...             try:
12  ...                 method = getattr(instance, action)
13  ...             except AttributeError:
14  ...                 raise exc.HTTPNotFound("No action %s" % action)
15  ...             resp = method()
16  ...             if isinstance(resp, basestring):
17  ...                 resp = Response(body=resp)
18  ...         except exc.HTTPException, e:
19  ...             resp = e
20  ...         return resp(environ, start_response)
21  ...     return replacement
```

line 1: Here we're kind of decorating a class. But really we'll just create a WSGI application wrapper.

line 2-4: The replacement WSGI application, also a closure. And we create a request and catch exceptions, just like in the decorator.

line 5: We instantiate the class with both the request and `req.urlvars` to initialize it. The instance will only be used for one request. (Note that the *instance* then doesn't have to be thread safe.)

line 6: We get the action variable out, if there is one.

line 7, 8: If there was one, we'll use the method name `{action}_{method}`...

line 9, 10: ... otherwise we'll use just the method for the method name.

line 11-14: We'll get the method from the instance, or respond with a 404 error if there is not such method.

line 15: Call the method, get the response

line 16, 17: If the response is just a string, create a full response object from it.

line 20: and then we forward the request...

line 21: ... and return the wrapper object we've created.

Here's the hello world:

```
>>> class Hello(object):
...     def __init__(self, req):
...         self.request = req
...     def get(self):
...         return '''<form method="POST">
...             You're name: <input type="text" name="name">
...             <input type="submit">
...             </form>'''
...     def post(self):
...         return 'Hello %s!' % self.request.params['name']
>>> hello = rest_controller(Hello)
```

We'll run the same test as before:

```
>>> hello_world = Router()
>>> hello_world.add_route('/', controller=hello)
>>> req = Request.blank('/')
>>> resp = req.get_response(hello_world)
>>> print resp
200 OK
Content-Type: text/html; charset=UTF-8
Content-Length: 131

<form method="POST">
            You're name: <input type="text" name="name">
            <input type="submit">
            </form>
>>> req.method = 'POST'
>>> req.body = 'name=Ian'
>>> resp = req.get_response(hello_world)
>>> print resp
200 OK
Content-Type: text/html; charset=UTF-8
Content-Length: 10

Hello Ian!
```

9.5.10 URL Generation and Request Access

You can use hard-coded links in your HTML, but this can have problems. Relative links are hard to manage, and absolute links presume that your application lives at a particular location. WSGI gives a variable `SCRIPT_NAME`, which is the portion of the path that led up to this application. If you are writing a blog application, for instance, someone might want to install it at `/blog/`, and then SCRIPT_NAME would be `"/blog"`. We should generate links with that in mind.

The base URL using SCRIPT_NAME is `req.application_url`. So, if we have access to the request we can make a URL. But what if we don't have access?

We can use thread-local variables to make it easy for any function to get access to the currect request. A "thread-local" variable is a variable whose value is tracked separately for each thread, so if there are multiple requests in different threads, their requests won't clobber each other.

The basic means of using a thread-local variable is `threading.local()`. This creates a blank object that can have thread-local attributes assigned to it. I find the best way to get *at* a thread-local value is with a function, as this makes it clear that you are fetching the object, as opposed to getting at some global object.

Here's the basic structure for the local:

```
>>> import threading
>>> class Localized(object):
...     def __init__(self):
...         self.local = threading.local()
...     def register(self, object):
...         self.local.object = object
...     def unregister(self):
...         del self.local.object
...     def __call__(self):
...         try:
...             return self.local.object
...         except AttributeError:
...             raise TypeError("No object has been registered for this thread")
>>> get_request = Localized()
```

Now we need some *middleware* to register the request object. Middleware is something that wraps an application, possibly modifying the request on the way in or the way out. In a sense the `Router` object was middleware, though not exactly because it didn't wrap a single application.

This registration middleware looks like:

```
>>> class RegisterRequest(object):
...     def __init__(self, app):
...         self.app = app
...     def __call__(self, environ, start_response):
...         req = Request(environ)
...         get_request.register(req)
...         try:
...             return self.app(environ, start_response)
...         finally:
...             get_request.unregister()
```

Now if we do:

```
>>> hello_world = RegisterRequest(hello_world)
```

then the request will be registered each time. Now, lets create a URL generation function:

```
>>> import urllib
>>> def url(*segments, **vars):
...     base_url = get_request().application_url
...     path = '/'.join(str(s) for s in segments)
...     if not path.startswith('/'):
...         path = '/' + path
...     if vars:
...         path += '?' + urllib.urlencode(vars)
...     return base_url + path
```

Now, to test:

```
>>> get_request.register(Request.blank('http://localhost/'))
>>> url('article', 1)
'http://localhost/article/1'
>>> url('search', q='some query')
'http://localhost/search?q=some+query'
```

9.5.11 Templating

Well, we don't *really* need to factor templating into our framework. After all, you return a string from your controller, and you can figure out on your own how to get a rendered string from a template.

But we'll add a little helper, because I think it shows a clever trick.

We'll use Tempita for templating, mostly because it's very simplistic about how it does loading. The basic form is:

```
import tempita
template = tempita.HTMLTemplate.from_filename('some-file.html')
```

But we'll be implementing a function `render(template_name, **vars)` that will render the named template, treating it as a path *relative to the location of the render() call*. That's the trick.

To do that we use `sys._getframe`, which is a way to look at information in the calling scope. Generally this is frowned upon, but I think this case is justifiable.

We'll also let you pass an instantiated template in instead of a template name, which will be useful in places like a doctest where there aren't other files easily accessible.

```
>>> import os
>>> import tempita
>>> def render(template, **vars):
...     if isinstance(template, basestring):
...         caller_location = sys._getframe(1).f_globals['__file__']
...         filename = os.path.join(os.path.dirname(caller_location), template)
...         template = tempita.HTMLTemplate.from_filename(filename)
...     vars.setdefault('request', get_request())
...     return template.substitute(vars)
```

9.5.12 Conclusion

Well, that's a framework. Ta-da!

Of course, this doesn't deal with some other stuff. In particular:

- Configuration
- Making your routes debuggable

- Exception catching and other basic infrastructure
- Database connections
- Form handling
- Authentication

But, for now, that's outside the scope of this document.

CHAPTER 10

Change History

10.1 What's New in WebOb 1.5

10.1.1 Backwards Incompatibilities

- `Response.set_cookie` renamed the only required parameter from "key" to "name". The code will now still accept "key" as a keyword argument, and will issue a DeprecationWarning until WebOb 1.7.
- The `status` attribute of a `Response` object no longer takes a string like `None None` and allows that to be set as the status. It now has to at least match the pattern of `<integer status code> <explenation of status code>`. Invalid status strings will now raise a `ValueError`.
- `Morsel` will no longer accept a cookie value that does not meet RFC6265's cookie-octet specification. Upon calling `Morsel.serialize` a warning will be issued, in the future this will raise a `ValueError`, please update your cookie handling code. See https://github.com/Pylons/webob/pull/172

  The cookie-octet specification in RFC6265 states the following characters are valid in a cookie value:

Hex Range	Actual Characters
[0x21]	!
[0x25-0x2B]	#$%&'()*+
[0x2D-0x3A]	-./0123456789:
[0x3C-0x5B]	<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[
[0x5D-0x7E]	]^_`abcdefghijklmnopqrstuvwxyz{|}~

  RFC6265 suggests using base 64 to serialize data before storing data in a cookie.

  Cookies that meet the RFC6265 standard will no longer be quoted, as this is unnecessary. This is a no-op as far as browsers and cookie storage is concerned.
- `Response.set_cookie` now uses the internal `make_cookie` API, which will issue warnings if cookies are set with invalid bytes. See https://github.com/Pylons/webob/pull/172

10.1.2 Features

- HTTP Status Code 308 is now supported as a Permanent Redirect. See https://github.com/Pylons/webob/pull/207
- Add support for some new caching headers, stale-while-revalidate and stale-if-error that can be used by reverse proxies to cache stale responses temporarily if the backend disappears. From RFC5861. See https://github.com/Pylons/webob/pull/189

10.1.3 Bug Fixes

- The exceptions HTTPNotAcceptable, HTTPUnsupportedMediaType and HTTPNotImplemented will now correctly use the sub-classed template rather than the default error template. See https://github.com/Pylons/webob/issues/221
- Response's from_file now correctly deals with a status line that contains an HTTP version identifier. HTTP/1.1 200 OK is now correctly parsed, whereas before this would raise an error upon setting the Response.status in from_file. See https://github.com/Pylons/webob/issues/121
- The cookie API functions will now make sure that *max_age* is an integer or an string that can convert to an integer. Previously passing in max_age='test' would have silently done the wrong thing.
- Unbreak req.POST when the request method is PATCH. Instead of returning something cmpletely unrelated we return NoVar. See: https://github.com/Pylons/webob/pull/215
- Response.status now uses duck-typing for integers, and has also learned to raise a ValueError if the status isn't an integer followed by a space, and then the reason. See https://github.com/Pylons/webob/pull/191
- Fixed a bug in `webob.multidict.GetDict` which resulted in the QUERY_STRING not being updated when changes were made to query params using `Request.GET.extend()`.
- Read the body of a request if we think it might have a body. This fixes PATCH to support bodies. See https://github.com/Pylons/webob/pull/184
- Response.from_file returns HTTP headers as latin1 rather than UTF-8, this fixes the usage on Google AppEngine. See https://github.com/Pylons/webob/issues/99 and https://github.com/Pylons/webob/pull/150
- Fix a bug in parsing the auth parameters that contained bad white space. This makes the parsing fall in line with what's required in RFC7235. See https://github.com/Pylons/webob/issues/158
- Use 'rn' line endings in `Response.__str__`. See: https://github.com/Pylons/webob/pull/146

10.1.4 Documentation Changes

- `response.set_cookie` now has proper documentation for `max_age` and `expires`. The code has also been refactored to use `cookies.make_cookie` instead of duplicating the code. This fixes https://github.com/Pylons/webob/issues/166 and https://github.com/Pylons/webob/issues/171
- Documentation didn't match the actual code for the wsgify function signature. See https://github.com/Pylons/webob/pull/167
- Remove the WebDAV only from certain HTTP Exceptions, these exceptions may also be used by REST services for example.

10.2 WebOb Change History

10.2.1 1.5.1 (2015-10-30)

Bug Fixes

- The exceptions HTTPNotAcceptable, HTTPUnsupportedMediaType and HTTPNotImplemented will now correctly use the sub-classed template rather than the default error template. See https://github.com/Pylons/webob/issues/221

- Response's from_file now correctly deals with a status line that contains an HTTP version identifier. HTTP/1.1 200 OK is now correctly parsed, whereas before this would raise an error upon setting the Response.status in from_file. See https://github.com/Pylons/webob/issues/121

10.2.2 1.5.0 (2015-10-11)

Bug Fixes

- The cookie API functions will now make sure that *max_age* is an integer or an string that can convert to an integer. Previously passing in max_age='test' would have silently done the wrong thing.

10.2.3 1.5.0b0 (2015-09-06)

Bug Fixes

- Unbreak req.POST when the request method is PATCH. Instead of returning something cmpletely unrelated we return NoVar. See: https://github.com/Pylons/webob/pull/215

Features

- HTTP Status Code 308 is now supported as a Permanent Redirect. See https://github.com/Pylons/webob/pull/207

10.2.4 1.5.0a1 (2015-07-30)

Backwards Incompatibilities

- `Response.set_cookie` renamed the only required parameter from "key" to "name". The code will now still accept "key" as a keyword argument, and will issue a DeprecationWarning until WebOb 1.7.
- The `status` attribute of a `Response` object no longer takes a string like `None None` and allows that to be set as the status. It now has to at least match the pattern of `<integer status code> <explenation of status code>`. Invalid status strings will now raise a `ValueError`.

10.2.5 1.5.0a0 (2015-07-25)

Backwards Incompatibilities

- `Morsel` will no longer accept a cookie value that does not meet RFC6265's cookie-octet specification. Upon calling `Morsel.serialize` a warning will be issued, in the future this will raise a `ValueError`, please update your cookie handling code. See https://github.com/Pylons/webob/pull/172

  The cookie-octet specification in RFC6265 states the following characters are valid in a cookie value:

Hex Range	Actual Characters
[0x21]	!
[0x25-0x2B]	#$%&'()*+
[0x2D-0x3A]	-./0123456789:
[0x3C-0x5B]	<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[
[0x5D-0x7E]	]^_`abcdefghijklmnopqrstuvwxyz{|}~

RFC6265 suggests using base 64 to serialize data before storing data in a cookie.

Cookies that meet the RFC6265 standard will no longer be quoted, as this is unnecessary. This is a no-op as far as browsers and cookie storage is concerned.

- `Response.set_cookie` now uses the internal `make_cookie` API, which will issue warnings if cookies are set with invalid bytes. See https://github.com/Pylons/webob/pull/172

Features

- Add support for some new caching headers, stale-while-revalidate and stale-if-error that can be used by reverse proxies to cache stale responses temporarily if the backend disappears. From RFC5861. See https://github.com/Pylons/webob/pull/189

Bug Fixes

- Response.status now uses duck-typing for integers, and has also learned to raise a ValueError if the status isn't an integer followed by a space, and then the reason. See https://github.com/Pylons/webob/pull/191
- Fixed a bug in `webob.multidict.GetDict` which resulted in the QUERY_STRING not being updated when changes were made to query params using `Request.GET.extend()`.
- Read the body of a request if we think it might have a body. This fixes PATCH to support bodies. See https://github.com/Pylons/webob/pull/184
- Response.from_file returns HTTP headers as latin1 rather than UTF-8, this fixes the usage on Google AppEngine. See https://github.com/Pylons/webob/issues/99 and https://github.com/Pylons/webob/pull/150
- Fix a bug in parsing the auth parameters that contained bad white space. This makes the parsing fall in line with what's required in RFC7235. See https://github.com/Pylons/webob/issues/158
- Use 'rn' line endings in `Response.__str__`. See: https://github.com/Pylons/webob/pull/146

Documentation Changes

- `response.set_cookie` now has proper documentation for `max_age` and `expires`. The code has also been refactored to use `cookies.make_cookie` instead of duplicating the code. This fixes https://github.com/Pylons/webob/issues/166 and https://github.com/Pylons/webob/issues/171
- Documentation didn't match the actual code for the wsgify function signature. See https://github.com/Pylons/webob/pull/167
- Remove the WebDAV only from certain HTTP Exceptions, these exceptions may also be used by REST services for example.

10.2.6 1.4 (2014-05-14)

Features

- Remove `webob.__version__`, the version number had not been kept in sync with the official pkg version. To obtain the WebOb version number, use `pkg_resources.get_distribution('webob').version` instead.

Bug Fixes

- Fix a bug in `EmptyResponse` that prevents it from setting self.close as appropriate due to testing truthiness of object rather than if it is something other than `None`.
- Fix a bug in `SignedSerializer` preventing secrets from containing higher-order characters. See https://github.com/Pylons/webob/issues/136
- Use the `hmac.compare_digest` method when available for constant-time comparisons.

10.2.7 1.3.1 (2013-12-13)

Bug Fixes

- Fix a bug in `SignedCookieProfile` whereby we didn't keep the original serializer around, this would cause us to have `SignedSerializer` be added on top of a `SignedSerializer` which would cause it to be run twice when attempting to verify a cookie. See https://github.com/Pylons/webob/pull/127

Backwards Incompatibilities

- When `CookieProfile.get_value` and `SignedCookieProfile.get_value` fails to deserialize a badly encoded value, we now return `None` as if the cookie was never set in the first place instead of allowing a `ValueError` to be raised to the calling code. See https://github.com/Pylons/webob/pull/126

10.2.8 1.3 (2013-12-10)

Features

- Added a read-only `domain` property to `BaseRequest`. This property returns the domain portion of the host value. For example, if the environment contains an `HTTP_HOST` value of `foo.example.com:8000`, `request.domain` will return `foo.example.com`.
- Added five new APIs: `webob.cookies.CookieProfile`, `webob.cookies.SignedCookieProfile`, `webob.cookies.JSONSerializer` and `webob.cookies.SignedSerializer`, and `webob.cookies.make_cookie`. These APIs are convenience APIs for generating and parsing cookie headers as well as dealing with signing cookies.
- Cookies generated via webob.cookies quoted characters in cookie values that did not need to be quoted per RFC 6265. The following characters are no longer quoted in cookie values: `~/=<>()[]{}?@` . The full set of non-letter-or-digit unquoted cookie value characters is now `!#$%&'*+-.^_`|~/: =<>()[]{}?@`. See http://tools.ietf.org/html/rfc6265#section-4.1.1 for more information.
- Cookie names are now restricted to the set of characters expected by RFC 6265. Previously they could contain unsupported characters such as `/`.
- Older versions of Webob escaped the doublequote to `\"` and the backslash to `\\` when quoting cookie values. Now, instead, cookie serialization generates `\042` for the doublequote and `\134` for the backslash. This is what is expected as per RFC 6265. Note that old cookie values that do have the older style quoting in them will still be unquoted correctly, however.
- Added support for draft status code 451 ("Unavailable for Legal Reasons"). See http://tools.ietf.org/html/draft-tbray-http-legally-restricted-status-00
- Added status codes 428, 429, 431 and 511 to `util.status_reasons` (they were already present in a previous release as `webob.exc` exceptions).

Bug Fixes

- MIMEAccept happily parsed malformed wildcard strings like "image/pn*" at parse time, but then threw an AssertionError during matching. See https://github.com/Pylons/webob/pull/83 .
- Preserve document ordering of GET and POST request data when POST data passed to Request.blank is a MultiDict. See https://github.com/Pylons/webob/pull/96
- Allow query strings attached to PATCH requests to populate request.params. See https://github.com/Pylons/webob/pull/106
- Added Python 3.3 trove classifier.

10.2.9 1.2.3

- Maintainership transferred to *Pylons Project <http://www.pylonsproject.org/>*
- Fix parsing of form submissions where fields have transfer-content-encoding headers.

10.2.10 1.2.2

- Fix multiple calls to `cache_expires()` not fully overriding the previously set headers.
- Fix parsing of form submissions where fields have different encodings.

10.2.11 1.2.1

- Add index page (e.g., `index.html`) support for *webob.static.DirectoryApp*.
- Detect mime-type when creating a test request with file uploads (`Request.blank("/", POST=dict(file1=("foo.jpg", "xxx")))`)
- Relax parsing of `Accept` and `Range` headers to allow uppercase and extra whitespace.
- Fix docs references to some deprecated classes.

10.2.12 1.2

- Fix *webob.client* handling of connection-refused on Windows.
- Use `simplejson` in *webob.request* if present.
- Fix `resp.retry_after = <long>` interpreting value as a UNIX timestamp (should interpret as time delta in seconds).

10.2.13 1.2rc1

- Add `Response.json` and `Request.json` which reads and sets the body using a JSON encoding (previously only the readable attribute `Request.json_body` existed). `Request.json_body` is still available as an alias.
- Rename `Response.status_int` to `Response.status_code` (the `.status_int` name is still available and will be supported indefinitely).
- Add `Request.text`, the unicode version of the request body (similar to `Response.text`).

- Add *webob.client* which contains the WSGI application `send_request_app` and `SendRequest`. All requests sent to this application are turned into HTTP requests.
- Renamed `Request.get_response(app)` to `Request.send(app)`. The `.get_response()` name is still available.
- Use `send_request_app` as the default application for `Request.send()`, so you can do:

```
resp = Request.blank("http://python.org").send()
```

- Add *webob.static* which contains two new WSGI applications, `FileApp` serve one static file and `DirectoryApp` to serve the content of a directory. They should provide a reusable implementation of WebOb File-Serving Example. It also comes with support for `wsgi.file_wrapper`.

  The implementation has been imported and simplified from `PasteOb.fileapp`.
- Add `dev` and `docs` setup.py aliases (to install development and docs dependencies respectively, e.g. "python setup.py dev").

10.2.14 1.2b3

- Added `request.host_port` API (returns port number implied by HTTP_HOST, falling back to SERVER_PORT).
- Added `request.client_addr` API (returns IP address implied by HTTP_X_FORWARDED_FOR, falling back to REMOTE_ADDR).
- Fix corner-case `response.status_int` and `response.status` mutation bug on py3 (use explicit floor division).
- Backwards incompatibility: Request and BaseRequest objects now return Unicode for `request.path_info` and `request.script_name` under Python 2. Rationale: the legacy behavior of returning the respective raw environ values was nonsensical on Python 3. Working with non-ascii encoded environ variables as raw WSGI values under Python 3 makes no sense, as PEP 3333 specifies that environ variables are bytes-tunneled-as-latin-1 strings.

  If you don't care about Python 3, and you need strict backwards compatibility, to get legacy behavior of returning bytes on Python 2 for these attributes, use `webob.LegacyRequest` instead of `webob.Request`. Although it's possible to use `webob.LegacyRequest` under Python 3, it makes no sense, and it should not be used there.
- The above backwards incompatibility fixed nonsensical behavior of `request.host_url`, `request.application_url`, `request.path_url`, `request.path`, `request.path_qs`, `request.url`, `request.relative_url`, `request.path_info_peek`, `request.path_info_pop` under Python 3. These methods previously dealt with raw SCRIPT_NAME and PATH_INFO values, which caused nonsensical results.
- The WebOb Request object now respects an additional WSGI environment variable: `webob.url_encoding`. `webob.url_encoding` will be used to decode the raw WSGI PATH_INFO and SCRIPT_NAME variables when the `request.path_info` and `request.script_name` APIs are used.
- Request objects now accept an additional constructor parameter: `url_encoding`. `url_encoding` will be used to decode PATH_INFO and SCRIPT_NAME from its WSGI-encoded values. If `webob.url_encoding` is not set in the environ and `url_encoding` is not passed to the Request constructor, the default value `utf-8` will be used to decode the PATH_INFO and SCRIPT_NAME.

  Note that passing `url_encoding` will cause the WSGI environment variable `webob.url_encoding` to be set.
- Fix `webob.response._request_uri` internal function to generate sensible request URI under Python 3. This fixed a problem under Python 3 if you were using non-absolute Location headers in responses.

10.2.15 1.2b2

- Fix `request.cookies.get('name', 'default')`. Previously `default` was ignored.

10.2.16 1.2b1

- Mutating the `request.cookies` property now reflects the mutations into the `HTTP_COOKIES` environ header.
- `Response.etag = (tag, False)` sets weak etag.
- `Range` only parses single range now.
- `Range.satisfiable(..)` is gone.
- `Accept.best_matches()` is gone; use `list(request.accept)` or `request.accept.best_match(..)` instead (applies to all Accept-* headers) or similar with `request.accept_language`.
- `Response.request` and `Response.environ` attrs are undeprecated and no longer raise exceptions when used. These can also be passed to the Response constructor. This is to support codebases that pass them to the constructor or assign them to a response instance. However, some behavior differences from 1.1 exist. In particular, synchronization is no longer done between environ and request attribute properties of Response; you may pass either to the constructor (or both) or assign one or the other or both, but they wont be managed specially and will remain the same over the lifetime of the response just as you passed them. Default values for both `request` and `environ` on any given response are `None` now.
- Undeprecated `uscript_name` and `upath_info`.
- For backwards compatibility purposes, switch `req.script_name` and `path_info` back again to contain "raw" undecoded native strings rather than text. Use `uscript_name` and `upath_info` to get the text version of SCRIPT_NAME and PATH_INFO.
- Don't raise an exception if `unicode_errors` or `decode_param_names` is passed to the Request constructor. Instead, emit a warning. For benefit of Pylons 1.X, which passes both.
- Don't raise an exception if HTTPException.exception is used; instead emit a warning. For benefit of Pylons 1.X, which uses it.

10.2.17 1.2a2

- `req.script_name` and `path_info` now contain text, not bytes.
- Deprecated `uscript_name` and `upath_info`.
- `charset` argument to `Request` as well as the attribute can only be set to UTF-8 or the value already present in the `Content-Type` header.
- `unicode_errors` attribute of `Request` and related functionality is gone.
- To process requests that come in an encoding different from UTF-8, the request needs to be transcoded like this: `req = req.decode('windows-1251')`
- Added support for weak ETag matching in conditional responses.
- Most of etag-related functionality was refactored.

10.2.18 1.2a1

- Python 3.2 compatibility.
- No longer compatible with Python 2.5 (only 2.6, 2.7, and 3.2 are supported).
- Switched VCS from Mercurial to Git
- Moved development to GitHub
- Added full history from PyCon 2011 sprint to the repository
- Change `LimitedLengthFile` and `FakeCGIBody` to inherit from `io.RawIOBase` and benefit from `io.BufferedReader`.
- Do not set `resp.request` in `req.get_response(app)`
- `Response.request` and `.environ` attrs are deprecated and raise exceptions when used.
- Deprecated request attributes `str_GET`, `str_POST`, `str_cookies` and `str_params` now raise exceptions when touched.
- Remove testing dependency on WebTest.
- Remove UnicodeMultiDict class; the result of `Request.GET` and `Request.POST` is now just a plain `MultiDict`.
- The `decode_param_names` Request constructor argument has been removed, along with the `Request.decode_param_names` attribute.
- The `Request.as_string()` method is now better known as `Request.as_bytes()`.
- The `Request.from_string()` method is now better known as `Request.from_bytes()`.
- A new method named `Request.as_text()` now exists.
- A new method named `Request.from_text()` now exists.
- The `webob.dec.wsgify` repr() is now much less informative, but a lot easier to test and maintain.

10.2.19 1.1.1

- Fix disconnect detection being incorrect in some cases (issue 21).
- Fix exception when calling `.accept.best_match(..)` on a header containing `'*'` (instead of `'*/*'`).
- Extract some of the `Accept` code into subclasses (`AcceptCharset`, `AcceptLanguage`).
- Improve language matching so that the app can now offer a generic language code and it will match any of the accepted dialects (`'en' in AcceptLanguage('en-gb')`).
- Normalize locale names when matching (`'en_GB' in AcceptLanguage('en-gb')`).
- Deprecate `etag.weak_match(..)`.
- Deprecate `Response.request` and `Response.environ` attrs.

10.2.20 1.1

- Remove deprecation warnings for `unicode_body` and `ubody`.

10.2.21 1.1rc1

- Deprecate `Response.ubody` / `.unicode_body` in favor of new `.text` attribute (the old names will be removed in 1.3 or even later).
- Make `Response.write` much more efficient (issue 18).
- Make sure copying responses does not reset Content-Length or Content-MD5 of the original (and that of future copies).
- Change `del res.body` semantics so that it doesn't make the response invalid, but only removes the response body.
- Remove `Response._body` so the `_app_iter` is the only representation.

10.2.22 1.1b2

- Add detection for browser / user-agent disconnects. If the client disconnected before sending the entire request body (POST / PUT), `req.POST`, `req.body` and other related properties and methods will raise an exception. Previously this caused the application get a truncated request with no indication that it is incomplete.
- Make `Response.body_file` settable. This is now valid: `Response(body_file=open('foo.bin'), content_type=...)`
- Revert the restriction on req.body not being settable for GET and some other requests. Such requests actually can have a body according to HTTP BIS (see also commit message)
- Add support for file upload testing via `Request.blank(POST=..)`. Patch contributed by Tim Perevezentsev. See also: ticket, changeset.
- Deprecate `req.str_GET`, `str_POST`, `str_params` and `str_cookies` (warning).
- Deprecate `req.decode_param_names` (warning).
- Change `req.decode_param_names` default to `True`. This means that `.POST`, `.GET`, `.params` and `.cookies` keys are now unicode. This is necessary for WebOb to behave as close as possible on Python 2 and Python 3.

10.2.23 1.1b1

- We have acquired the webob.org domain, docs are now hosted at docs.webob.org
- Make `accept.quality(..)` return best match quality, not first match quality.
- Fix `Range.satisfiable(..)` edge cases.
- Make sure `WSGIHTTPException` instances return the same headers for `HEAD` and `GET` requests.
- Drop Python 2.4 support
- Deprecate `HTTPException.exception` (warning on use).
- Deprecate `accept.first_match(..)` (warning on use). Use `.best_match(..)` instead.
- Complete deprecation of `req.[str_]{post|query}vars` properties (exception on use).
- Remove `FakeCGIBody.seek` hack (no longer necessary).

10.2.24 1.0.8

- Escape commas in cookie values (see also: stdlib Cookie bug)
- Change cookie serialization to more closely match how cookies usually are serialized (unquoted expires, semicolon separators even between morsels)
- Fix some rare cases in cookie parsing
- Enhance the req.is_body_readable to always guess GET, HEAD, DELETE and TRACE as unreadable and PUT and POST as readable (issue 12)
- Deny setting req.body or req.body_file to non-empty values for GET, HEAD and other bodiless requests
- Fix running nosetests with arguments on UNIX systems (issue 11)

10.2.25 1.0.7

- Fix `Accept` header matching for items with zero-quality (issue 10)
- Hide password values in `MultiDict.__repr__`

10.2.26 1.0.6

- Use `environ['wsgi.input'].read()` instead of `.read(-1)` because the former is explicitly mentioned in PEP-3333 and CherryPy server does not support the latter.
- Add new `environ['webob.is_body_readable']` flag which specifies if the input stream is readable even if the `CONTENT_LENGTH` is not set. WebOb now only ever reads the input stream if the content-length is known or this flag is set.
- The two changes above fix a hangup with CherryPy and wsgiref servers (issue 6)
- `req.body_file` is now safer to read directly. For `GET` and other similar requests it returns an empty `StringIO` or `BytesIO` object even if the server passed in something else.
- Setting `req.body_file` to a string now produces a PendingDeprecationWarning. It will produce DeprecationWarning in 1.1 and raise an error in 1.2. Either set `req.body_file` to a file-like object or set `req.body` to a string value.
- Fix `.pop()` and `.setdefault(..)` methods of `req/resp.cache_control`
- Thanks to the participants of Pyramid sprint at the PyCon US 2011 WebOb now has 100% test coverage.

10.2.27 1.0.5

- Restore Python 2.4 compatibility.

10.2.28 1.0.4

- The field names escaping bug semi-fixed in 1.0.3 and originally blamed on cgi module was in fact a `webob.request._encode_multipart` bug (also in Google Chrome) and was lurking in webob code for quite some time – 1.0.2 just made it trigger more often. Now it is fixed properly.
- Make sure that req.url and related properties do not unnecessarily escape some chars (`:@&+$`) in the URI path (issue 5)

- Revert some changes from 1.0.3 that have broken backwards compatibility for some apps. Getting `req.body_file` does not make input stream seekable, but there's a new property `req.body_file_seekable` that does.
- `Request.get_response` and `Request.call_application` seek the input body to start before calling the app (if possible).
- Accessing `req.body` 'rewinds' the input stream back to pos 0 as well.
- When accessing `req.POST` we now avoid making the body seekable as the input stream data are preserved in `FakeCGIBody` anyway.
- Add new method `Request.from_string`.
- Make sure `Request.as_string()` uses CRLF to separate headers.
- Improve parity between `Request.as_string()` and `.from_file`/`.from_string` methods, so that the latter can parse output of the former and create a similar request object which wasn't always the case previously.

10.2.29 1.0.3

- Correct a caching issue introduced in WebOb 1.0.2 that was causing unnecessary reparsing of POST requests.
- Fix a bug regarding field names escaping for forms submitted as `multipart/form-data`. For more infromation see the bug report and discussion and 1.0.4 notes for further fix.
- Add `req.http_version` attribute.

10.2.30 1.0.2

- Primary maintainer is now Sergey Schetinin.
- Issue tracker moved from Trac to bitbucket's issue tracker
- WebOb 1.0.1 changed the behavior of `MultiDict.update` to be more in line with other dict-like objects. We now also issue a warning when we detect that the client code seems to expect the old, extending semantics.
- Make `Response.set_cookie(key, None)` set the 'delete-cookie' (same as `.delete_cookie(key)`)
- Make `req.upath_info` and `req.uscript_name` settable
- Add :meth:`Request.as_string()` method
- Add a `req.is_body_seekable` property
- Support for the `deflate` method with `resp.decode_content()`
- To better conform to WSGI spec we no longer attempt to use seek on `wsgi.input` file instead we assume it is not seekable unless `env['webob.is_body_seekable']` is set. When making the body seekable we set that flag.
- A call to `req.make_body_seekable()` now guarantees that the body is seekable, is at 0 position and that a correct `req.content_length` is present.
- `req.body_file` is always seekable. To access `env['wsgi.input']` without any processing, use `req.body_file_raw`. (Partially reverted in 1.0.4)
- Fix responses to HEAD requests with Range.
- Fix `del resp.content_type`, `del req.body`, `del req.cache_control`
- Fix `resp.merge_cookies()` when called with an argument that is not a Response instance.

- Fix `resp.content_body = None` (was removing Cache-Control instead)
- Fix `req.body_file = f` setting `CONTENT_LENGTH` to `-1` (now removes from environ)
- Fix: make sure req.copy() leaves the original with seekable body
- Fix handling of WSGI environs with missing `SCRIPT_NAME`
- A lot of tests were added by Mariano Mara and Danny Navarro.

10.2.31 1.0.1

- As WebOb requires Python 2.4 or later, drop some compatibility modules and update the code to use the decorator syntax.
- Implement optional on-the-fly response compression (`resp.encode_content(lazy=True)`)
- Drop `util.safezip` module and make `util` a module instead of a subpackage. Merge `statusreasons` into it.
- Instead of using stdlib `Cookie` with monkeypatching, add a derived but thoroughly rewritten, cleaner, safer and faster `webob.cookies` module.
- Fix: `Response.merge_cookies` now copies the headers before modification instead of doing it in-place.
- Fix: setting request header attribute to `None` deletes that header. (Bug only affected the 1.0 release).
- Use `io.BytesIO` for the request body file on Python 2.7 and newer.
- If a UnicodeMultiDict was used as the `multi` argument of another UnicodeMultiDict, and a `cgi.FieldStorage` with a `filename` with high-order characters was present in the underlying UnicodeMultiDict, a `UnicodeEncodeError` would be raised when any helper method caused the `_decode_value` method to be called, because the method would try to decode an already decoded string.
- Fix tests to pass under Python 2.4.
- Add descriptive docstrings to each exception in `webob.exc`.
- Change the behaviour of `MultiDict.update` to overwrite existing header values instead of adding new headers. The extending semantics are now available via the `extend` method.
- Fix a bug in `webob.exc.WSGIHTTPException.__init__`. If a list of `headers` was passed as a sequence which contained duplicate keys (for example, multiple `Set-Cookie` headers), all but one of those headers would be lost, because the list was effectively flattened into a dictionary as the result of calling `self.headers.update`. Fixed via calling `self.headers.extend` instead.

10.2.32 1.0

- 1.0, yay!
- Pull in werkzeug Cookie fix for malformed cookie bug.
- Implement `Request.from_file()` and `Response.from_file()` which are kind of the inversion of `str(req)` and `str(resp)`
- Add optional `pattern` argument to `Request.path_info_pop()` that requires the `path_info` segment to match the passed regexp to get popped and returned.
- Rewrite most of descriptor implementations for speed.
- Reorder descriptor declarations to group them by their semantics.
- Move code around so that there are fewer compat modules.

- Change :meth:`HTTPError.__str__` to better conform to PEP 352.
- Make `Request.cache_control` a view on the headers.
- Correct Accept-Language and Accept-Charset matching to fully conform to the HTTP spec.
- Expose parts of `Request.blank()` as `environ_from_url()` and `environ_add_POST()`
- Fix Authorization header parsing for some corner cases.
- Fix an error generated if the user-agent sends a 'Content_Length' header (note the underscore).
- Kill `Request.default_charset`. Request charset defaults to UTF-8. This ensures that all values in `req.GET`, `req.POST` and `req.params` are always unicode.
- Fix the `headerlist` and `content_type` constructor arguments priorities for `HTTPError` and subclasses.
- Add support for weak etags to conditional Response objects.
- Fix locale-dependence for some cookie dates strings.
- Improve overall test coverage.
- Rename class `webob.datastruct.EnvironHeaders` to `webob.headers.EnvironHeaders`
- Rename class `webob.headerdict.HeaderDict` to `webob.headers.ResponseHeaders`
- Rename class `webob.updatedict.UpdateDict` to `webob.cachecontrol.UpdateDict`

10.2.33 0.9.8

- Fix issue with WSGIHTTPException inadvertently generating unicode body and failing to encode it
- WWW-Authenticate response header is accessible as `response.www_authenticate`
- `response.www_authenticate` and `request.authorization` hold None or tuple `(auth_method, params)` where `params` is a dictionary (or a string when `auth_method` is not one of known auth schemes and for Authenticate: Basic ...)
- Don't share response headers when getting a response like `resp = req.get_response(some_app)`; this can avoid some funny errors with modifying headers and reusing Response objects.
- Add *overwrite* argument to `Response.set_cookie()` that make the new value overwrite the previously set. *False* by default.
- Add *strict* argument to `Response.unset_cookie()` that controls if an exception should be raised in case there are no cookies to unset. *True* by default.
- Fix `req.GET.copy()`
- Make sure that 304 Not Modified responses generated by `Response.conditional_response_app()` exclude Content-{Length/Type} headers
- Fix `Response.copy()` not being an independent copy
- When the requested range is not satisfiable, return a 416 error (was returning entire body)
- Truncate response for range requests that go beyond the end of body (was treating as invalid).

10.2.34 0.9.7.1

- Fix an import problem with Pylons

10.2.35 0.9.7

- Moved repository from svn location to http://bitbucket.org/ianb/webob/
- Arguments to `Accept.best_match()` must be specific types, not wildcards. The server should know a list of specic types it can offer and use `best_match` to select a specific one.
- With `req.accept.best_match([types])` prefer the first type in the list (previously it preferred later types).
- Also, make sure that if the user-agent accepts multiple types and there are multiple matches to the types that the application offers, `req.accept.best_match([..])` returns the most specific match. So if the server can satisfy either `image/*` or `text/plain` types, the latter will be picked independent from the order the accepted or offered types are listed (given they have the same quality rating).
- Fix Range, Content-Range and AppIter support all of which were broken in many ways, incorrectly parsing ranges, reporting incorrect content-ranges, failing to generate the correct body to satisfy the range from `app_iter` etc.
- Fix assumption that presense of a `seek` method means that the stream is seekable.
- Add `ubody` alias for `Response.unicode_body`
- Add Unicode versions of `Request.script_name` and `path_info`: `uscript_name` and `upath_info`.
- Split __init__.py into four modules: request, response, descriptors and datetime_utils.
- Fix `Response.body` access resetting Content-Length to zero for HEAD responses.
- Support passing Unicode bodies to `WSGIHTTPException` constructors.
- Make `bool(req.accept)` return `False` for requests with missing Accept header.
- Add HTTP version to `Request.__str__()` output.
- Resolve deprecation warnings for parse_qsl on Python 2.6 and newer.
- Fix `Response.md5_etag()` setting Content-MD5 in incorrect format.
- Add `Request.authorization` property for Authorization header.
- Make sure ETag value is always quoted (required by RFC)
- Moved most `Request` behavior into a new class named `BaseRequest`. The `Request` class is now a superclass for `BaseRequest` and a simple mixin which manages `environ['webob.adhoc_attrs']` when `__setitem__`, `__delitem__` and `__getitem__` are called. This allows framework developers who do not want the `environ['webob.adhoc_attrs']` mutation behavior from `__setattr__`. (chrism)
- Added response attribute `response.content_disposition` for its associated header.
- Changed how `charset` is determined on `webob.Request` objects. Now the `charset` parameter is read on the Content-Type header, if it is present. Otherwise a `default_charset` parameter is read, or the `charset` argument to the Request constructor. This is more similar to how `webob.Response` handles the charset.
- Made the case of the Content-Type header consistent (note: this might break some doctests).
- Make `req.GET` settable, such that `req.environ['QUERY_STRING']` is updated.
- Fix problem with `req.POST` causing a re-parse of the body when you instantiate multiple `Request` objects over the same environ (e.g., when using middleware that looks at `req.POST`).
- Recreate the request body properly when a `POST` includes file uploads.
- When `req.POST` is updated, the generated body will include the new values.

- Added a `POST` parameter to `webob.Request.blank()`; when given this will create a request body for the POST parameters (list of two-tuples or dictionary-like object). Note: this does not handle unicode or file uploads.
- Added method `webob.Response.merge_cookies()`, which takes the `Set-Cookie` headers from a Response, and merges them with another response or WSGI application. (This is useful for flash messages.)
- Fix a problem with creating exceptions like `webob.exc.HTTPNotFound(body='<notfound/>', content_type='application/xml')` (i.e., non-HTML exceptions).
- When a Location header is not absolute in a Response, it will be made absolute when the Response is called as a WSGI application. This makes the response less bound to a specific request.
- Added *webob.dec*, a decorator for making WSGI applications from functions with the signature `resp = app(req)`.

10.2.36 0.9.6.1

- Fixed `Response.__init__()`, which for some content types would raise an exception.
- The `req.body` property will not recreate a StringIO object unnecessarily when rereading the body.

10.2.37 0.9.6

- Removed *environ_getter* from `webob.Request`. This largely-unused option allowed a Request object to be instantiated with a dynamic underlying environ. Since it wasn't used much, and might have been ill-advised from the beginning, and affected performance, it has been removed (from Chris McDonough).
- Speed ups for `webob.Response.__init__()` and `webob.Request.__init__()`
- Fix defaulting of `CONTENT_TYPE` instead of `CONTENT_LENGTH` to 0 in `Request.str_POST`.
- Added `webob.Response.copy()`

10.2.38 0.9.5

- Fix `Request.blank('/').copy()` raising an exception.
- Fix a potential memory leak with HEAD requests and 304 responses.
- Make *webob.html_escape()* respect the `.__html__()` magic method, which allows you to use HTML in *webob.exc.HTTPException* instances.
- Handle unicode values for `resp.location`.
- Allow arbitrary keyword arguments to `exc.HTTP*` (the same keywords you can send to `webob.Response`).
- Allow setting `webob.Response.cache_expires()` (usually it is called as a method). This is primarily to allow `Response(cache_expires=True)`.

10.2.39 0.9.4

- Quiet Python 2.6 deprecation warnings.
- Added an attribute `unicode_errors` to `webob.Response` – if set to something like `unicode_errors='replace'` it will decode `resp.body` appropriately. The default is `strict` (which was the former un-overridable behavior).

10.2.40 0.9.3

- Make sure that if changing the body the Content-MD5 header is removed. (Otherwise a lot of middleware would accidentally corrupt responses).
- Fixed `Response.encode_content('identity')` case (was a no-op even for encoded bodies).
- Fixed `Request.remove_conditional_headers()` that was removing If-Match header instead of If-None-Match.
- Fixed `resp.set_cookie(max_age=timedelta(...))`
- `request.POST` now supports PUT requests with the appropriate Content-Type.

10.2.41 0.9.2

- Add more arguments to `Request.remove_conditional_headers()` for more fine-grained control: *remove_encoding*, *remove_range*, *remove_match*, *remove_modified*. All of them are *True* by default.
- Add an *set_content_md5* argument to `Response.md5_etag()` that calculates and sets Content-MD5 reponse header from current body.
- Change formatting of cookie expires, to use the more traditional format `Wed, 5-May-2001 15:34:10 GMT` (dashes instead of spaces). Browsers should deal with either format, but some other code expects dashes.
- Added in `sorted` function for backward compatibility with Python 2.3.
- Allow keyword arguments to `webob.Request`, which assign attributes (possibly overwriting values in the environment).
- Added methods `webob.Request.make_body_seekable()` and `webob.Request.copy_body()`, which make it easier to share a request body among different consuming applications, doing something like *req.make_body_seekable(); req.body_file.seek(0)*

10.2.42 0.9.1

- `request.params.copy()` now returns a writable MultiDict (before it returned an unwritable object).
- There were several things broken with `UnicodeMultiDict` when `decode_param_names` is turned on (when the dictionary keys are unicode).
- You can pass keyword arguments to `Request.blank()` that will be used to construct `Request` (e.g., `Request.blank('/', decode_param_names=True)`).
- If you set headers like `response.etag` to a unicode value, they will be encoded as ISO-8859-1 (however, they will remain encoded, and `response.etag` will not be a unicode value).
- When parsing, interpret times with no timezone as UTC (previously they would be interpreted as local time).
- Set the Expires property on cookies when using `response.set_cookie()`. This is inherited from `max_age`.
- Support Unicode cookie values

10.2.43 0.9

- Added `req.urlarg`, which represents positional arguments in `environ['wsgiorg.routing_args']`.

- For Python 2.4, added attribute get/set proxies on exception objects from, for example, `webob.exc.HTTPNotFound().exception`, so that they act more like normal response objects (despite not being new-style classes or `webob.Response` objects). In Python 2.5 the exceptions are `webob.Response` objects.

Backward Incompatible Changes

- The `Response` constructor has changed: it is now `Response([body], [status], ...)` (before it was `Response([status], [body], ...)`). Body may be str or unicode.
- The `Response` class defaults to `text/html` for the Content-Type, and `utf8` for the charset (charset is only set on `text/*` and `application/*+xml` responses).

Bugfixes and Small Changes

- Use `BaseCookie` instead of `SimpleCookie` for parsing cookies.
- Added `resp.write(text)` method, which is equivalent to `resp.body += text` or `resp.unicode_body += text`, depending on the type of `text`.
- The `decode_param_names` argument (used like `Request(decode_param_names=True)`) was being ignored.
- Unicode decoding of file uploads and file upload filenames were causing errors when decoding non-file-upload fields (both fixes from Ryan Barrett).

10.2.44 0.8.5

- Added response methods `resp.encode_content()` and `resp.decode_content()` to gzip or ungzip content.
- `Response(status=404)` now works (before you would have to use `status="404 Not Found"`).
- Bugfix (typo) with reusing POST body.
- Added `226 IM Used` response status.
- Backport of `string.Template` included for Python 2.3 compatibility.

10.2.45 0.8.4

- `__setattr__` would keep `Request` subclasses from having properly settable environ proxies (like `req.path_info`).

10.2.46 0.8.3

- `request.POST` was giving FieldStorage objects for *every* attribute, not just file uploads. This is fixed now.
- Added request attributes `req.server_name` and `req.server_port` for the environ keys `SERVER_NAME` and `SERVER_PORT`.
- Avoid exceptions in `req.content_length`, even if `environ['CONTENT_LENGTH']` is somehow invalid.

10.2.47 0.8.2

- Python 2.3 compatibility: backport of `reversed(seq)`
- Made separate `.exception` attribute on `webob.exc` objects, since new-style classes can't be raised as exceptions.
- Deprecate `req.postvars` and `req.queryvars`, instead using the sole names `req.GET` and `req.POST` (also `req.str_GET` and `req.str_POST`). The old names give a warning; will give an error in next release, and be completely gone in the following release.
- `req.user_agent` is now just a simple string (parsing the User-Agent header was just too volatile, and required too much knowledge about current browsers). Similarly, `req.referer_search_query()` is gone.
- Added parameters `version` and `comment` to `Response.set_cookie()`, per William Dode's suggestion.
- Was accidentally consuming file uploads, instead of putting the `FieldStorage` object directly in the parameters.

10.2.48 0.8.1

- Added `res.set_cookie(..., httponly=True)` to set the `HttpOnly` attribute on the cookie, which keeps Javascript from reading the cookie.
- Added some WebDAV-related responses to `webob.exc`
- Set default `Last-Modified` when using `response.cache_expire()` (fixes issue with Opera)
- Generally fix `.cache_control`

10.2.49 0.8

First release. Nothing is new, or everything is new, depending on how you think about it.

CHAPTER 11

Status & License

WebOb is an extraction and refinement of pieces from Paste. It is under active development. Discussion should happen on the Pylons-discuss maillist, and bugs can go on the issue tracker. It was originally written by Ian Bicking, and is being maintained by the Pylons Project.

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

WebOb is released under an MIT-style license.

WebOb development happens on GitHub. Development version is installable via easy_install webob==dev. You can clone the source code with:

```
$ git clone https://github.com/Pylons/webob.git
```

a

c

d

e

m

r

s

Symbols

A

B

C

D

E

F

G

H

I

U

V

W