WebTest Documentation

Release 2.0

Ian Bicking

May 23, 2013

CONTENTS

author Ian Bicking <ianb@colorstudy.com>

maintainer Gael Pasgrimaud <gael@gawel.org>

CHAPTER

ONE

STATUS & LICENSE

WebTest is an extraction of `paste.fixture.TestApp`, rewriting portions to use WebOb. It is under active development as part of the Pylons cloud of packages.

Feedback and discussion should take place on the Pylons discuss list, and bugs should go into the Github tracker.

This library is licensed under an MIT-style license.

CHAPTER

TWO

INSTALLATION

You can use pip or easy_install to get the latest stable release:

```
$ pip install WebTest
$ easy_install WebTest
```

Or if you want the development version:

```
$ pip install https://nodeload.github.com/Pylons/webtest/tar.gz/master
```

CHAPTER

THREE

WHAT THIS DOES

WebTest helps you test your WSGI-based web applications. This can be any application that has a WSGI interface, including an application written in a framework that supports WSGI (which includes most actively developed Python web frameworks – almost anything that even nominally supports WSGI should be testable).

With this you can test your web applications without starting an HTTP server, and without poking into the web framework shortcutting pieces of your application that need to be tested. The tests WebTest runs are entirely equivalent to how a WSGI HTTP server would call an application. By testing the full stack of your application, the WebTest testing model is sometimes called a *functional test*, *integration test*, or *acceptance test* (though the latter two are not particularly good descriptions). This is in contrast to a *unit test* which tests a particular piece of functionality in your application. While complex programming tasks are often is suited to unit tests, template logic and simple web programming is often best done with functional tests; and regardless of the presence of unit tests, no testing strategy is complete without high-level tests to ensure the entire programming system works together.

WebTest helps you create tests by providing a convenient interface to run WSGI applications and verify the output.

CHAPTER

FOUR

QUICK START

The most important object in WebTest is `TestApp`, the wrapper for WSGI applications. To use it, you simply instantiate it with your WSGI application. (Note: if your WSGI application requires any configuration, you must set that up manually in your tests.)

The main class it `TestApp` which wrap your WSGI application and allow you to perform HTTP request on it.

Here is a basic application:

```
>>> def application(environ, start_response):
...     headers = [('Content-Type', 'text/html; charset=utf8'),
...                ('Content-Length', str(len(body)))]
...     start_response('200 Ok', headers)
...     return [body]
```

Wrap it into a `TestApp`:

```
>>> from webtest import TestApp
>>> app = TestApp(application)
```

Then you can get the response of a HTTP GET:

```
>>> resp = app.get('/')
```

And check the result. Like like reponse's status:

```
>>> assert resp.status == '200 Ok'
>>> assert resp.status_int == 200
```

Response's headers:

```
>>> assert resp.content_type == 'text/html'
>>> assert resp.content_length > 0
```

Or response's body:

```
>>> resp.mustcontain('<html>')
>>> assert 'form' in resp
```

WebTest can do much more. In particular it can handle *Form handling* and `json`.

CHAPTER

FIVE

CONTENTS

5.1 Functional Testing of Web Applications

5.1.1 TestApp

Making Requests

To make a request, use:

```
app.get('/path', [params], [headers], [extra_environ], ...)
```

This call to `get()` does a request for `/path`, with any params, extra headers or WSGI environment keys that you indicate. This returns a `TestResponse` object, based on `webob.response.Response`. It has some additional methods to make it easier to test.

If you want to do a POST request, use:

```
app.post('/path', {'vars': 'values'}, [headers], [extra_environ],
         [upload_files], ...)
```

Specifically the second argument of `post()` is the *body* of the request. You can pass in a dictionary (or dictionary-like object), or a string body (dictionary objects are turned into HTML form submissions).

You can also pass in the keyword argument upload_files, which is a list of `[(fieldname, filename, field_content)]`. File uploads use a different form submission data type to pass the structured data.

You can use `put()` and `delete()` for PUT and DELETE requests.

Making JSON Requests

Webtest provide some facilities to test json apis.

The `*_json` methods will transform data to json before `POST`/`PUT` and add the correct `Content-Type` for you.

Also Response have an attribute `.json` to allow you to retrieve json contents as a python dict.

Doing *POST* request with `webtest.TestApp.post_json()`:

```
>>> resp = app.post_json('/resource/', dict(id=1, value='value'))
>>> print(resp.request)
POST /resource/ HTTP/1.0
Content-Length: 27
Content-Type: application/json
...
```

```
>>> resp.json == {'id': 1, 'value': 'value'}
True
```

Doing *GET* request with `webtest.TestApp.get()` and using `webtest.response.json`:

To just parse body of the response, use Response.json:

```
>>> resp = app.get('/resource/1/')
>>> print(resp.request)
GET /resource/1/ HTTP/1.0
...

>>> resp.json == {'id': 1, 'value': 'value'}
True
```

Modifying the Environment & Simulating Authentication

The best way to simulate authentication is if your application looks in `environ['REMOTE_USER']` to see if someone is authenticated. Then you can simply set that value, like:

```
app.get('/secret', extra_environ=dict(REMOTE_USER='bob'))
```

If you want *all* your requests to have this key, do:

```
app = TestApp(my_app, extra_environ=dict(REMOTE_USER='bob'))
```

Testing a non wsgi application

You can use WebTest to test an application on a real web server. Just pass an url to the *TestApp* instead of a WSGI application:

```
app = TestApp('http://my.cool.websi.te')
```

You can also use the `WEBTEST_TARGET_URL` env var to switch from a WSGI application to a real server without having to modify your code:

```
os.environ['WEBTEST_TARGET_URL'] = 'http://my.cool.websi.te'
app = TestApp(wsgiapp) # will use the WEBTEST_TARGET_URL instead of the wsgiapp
```

By default the proxy will use `httplib` but you can use other backends by adding an anchor to your url:

```
app = TestApp('http://my.cool.websi.te#urllib3')
app = TestApp('http://my.cool.websi.te#requests')
app = TestApp('http://my.cool.websi.te#restkit')
```

What Is Tested By Default

A key concept behind WebTest is that there's lots of things you shouldn't have to check everytime you do a request. It is assumed that the response will either be a 2xx or 3xx response; if it isn't an exception will be raised (you can override this for a request, of course). The WSGI application is tested for WSGI compliance with a slightly modified version of wsgiref.validate (modified to support arguments to `InputWrapper.readline`) automatically. Also it checks that nothing is printed to the `environ['wsgi.errors']` error stream, which typically indicates a problem (one that would be non-fatal in a production situation, but if you are testing is something you should avoid).

To indicate another status is expected, use the keyword argument `status=404` to (for example) check that it is a 404 status, or `status="*"` to allow any status.

If you expect errors to be printed, use `expect_errors=True`.

5.1.2 TestResponse

The response object is based on `webob.response.Response` with some additions to help with testing.

The inherited attributes that are most interesting:

`response.status`: The text status of the response, e.g., `"200 OK"`.

`response.status_int`: The text status_int of the response, e.g., `200`.

`response.headers`: A dictionary-like object of the headers in the response.

`response.body`: The text body of the response.

`response.text`: The unicode text body of the response.

`response.normal_body`: The whitespace-normalized [1] body of the response.

`response.request`: The `webob.request.BaseRequest` object used to generate this response.

The added methods:

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

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

`response.mustcontain(string1, string2, no=string3)`: Raises an error if any of the strings are not found in the response. If a string of a string list is given as *no* keyword argument, raise an error if one of those are found in the response. It also prints out the response in that case, so you can see the real response.

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

`str(response)`: Gives a slightly-compacted version of the response. This is compacted to remove newlines, making it easier to use with doctest

`response.click(description=None, linkid=None, href=None, anchor=None, index=None, verbose=`
Clicks the described link (see `click`)

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

`response.form`: If there is just a single form, this returns that. It is an error if you use this and there are multiple forms.

Form handling

Getting a form

If you have a single html form in your page, just use the `.form` attribute:

```
>>> res = app.get('/form.html')
>>> form = res.form
```

You can use the form index if your html contains more than one form:

[1] The whitespace normalization replace sequences of whitespace characters and `\n \r \t` by a single space.

```
>>> form = res.forms[0]
```

Or the form id:

```
>>> form = res.forms['myform']
```

You can check form attributes:

```
>>> print(form.id)
myform
>>> print(form.action)
/form-submit
>>> print(form.method)
POST
```

Filling a form

You can fill out and submit forms from your tests. Fields are a dict like object:

```
>>> # dict of fields
>>> form.fields.values()
[(u'text', [<Text name="text">]), ..., (u'submit', [<Submit name="submit">])]
```

You can check the current value:

```
>>> print(form['text'].value)
Foo
```

Then you fill it in fields:

```
>>> form['text'] = 'Bar'
>>> # When names don't point to a single field:
>>> form.set('text', 'Bar', index=0)
```

Field types

Input and textarea fields

```
>>> print(form['textarea'].value)
Some text
>>> form['textarea'] = 'Some other text'
```

You can force the value of an hidden field:

```
>>> form['hidden'].force_value('2')
```

Select fields Simple select:

```
>>> print(form['select'].value)
option2
>>> form['select'] = 'option1'
```

Select multiple:

```
>>> print(form['multiple'].value)
['option2', 'option3']
>>> form['multiple'] = ['option1']
```

Select fields can only be set to valid values (i.e., values in an <option>) but you can also use `.force_value()` to enter values not present in an option.

```
>>> form['select'].force_value(['optionX'])
>>> form['multiple'].force_value(['optionX'])
```

Checkbox You can check if the checkbox is checked and is value:

```
>>> print(form['checkbox'].checked)
False
>>> print(form['checkbox'].value)
None
```

You can change the status with the value:

```
>>> form['checkbox'] = True
```

Or with the checked attribute:

```
>>> form['checkbox'].checked =True
```

If the checkbox is checked then you'll get the value:

```
>>> print(form['checkbox'].checked)
True
>>> print(form['checkbox'].value)
checkbox 1
```

If the checkbox has no value then it will be 'on' if you checked it:

```
>>> print(form['checkbox2'].value)
None
>>> form['checkbox2'].checked = True
>>> print(form['checkbox2'].value)
on
```

Radio

```
>>> print(form['radio'].value)
Radio 2
>>> form['radio'] = 'Radio 1'
```

File You can deal with file upload by using the Upload class:

```
>>> from webtest import Upload
>>> form['file'] = Upload('README.rst')
>>> form['file'] = Upload('README.rst', b'data')
```

Submit a form

Then you can submit the form:

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

```
>>> print(res)
Response: 200 OK
Content-Type: text/plain
text=Bar
...
submit=Submit
```

You can also use a specific submit button:

```
>>> res = form.submit('submit', index=1)
>>> print(res)
Response: 200 OK
Content-Type: text/plain
...
submit=Submit 2
```

Parsing the Body

There are several ways to get parsed versions of the response. These are the attributes:

response.html: Return a BeautifulSoup version of the response body:

```
>>> res = app.get('/index.html')
>>> res.html
<html><body><div id="content">hey!</div></body></html>
>>> res.html.__class__
<class '...BeautifulSoup'>
```

response.xml: Return an ElementTree version of the response body:

```
>>> res = app.get('/document.xml')
>>> res.xml
<Element 'xml' ...>
>>> res.xml[0].tag
'message'
>>> res.xml[0].text
'hey!'
```

response.lxml: Return an lxml version of the response body:

```
>>> res = app.get('/index.html')
>>> res.lxml
<Element html at ...>
>>> res.lxml.xpath('//body/div')[0].text
'hey!'

>>> res = app.get('/document.xml')
>>> res.lxml
<Element xml at ...>
>>> res.lxml[0].tag
'message'
>>> res.lxml[0].text
'hey!'
```

response.pyquery: Return an PyQuery version of the response body:

```
>>> res.pyquery('message')
[<message>]
```

```
>>> res.pyquery('message').text()
'hey!'
```

response.json: Return the parsed JSON (parsed with simplejson):

```
>>> res = app.get('/object.json')
>>> sorted(res.json.values())
[1, 2]
```

In each case the content-type must be correct or an AttributeError is raised. If you do not have the necessary library installed (none of them are required by WebTest), you will get an ImportError.

5.1.3 HTTP client/server utilities

This module contains some helpers to deal with the real http world.

class webtest.http.**StopableWSGIServer**(*application*, **args*, ***kwargs*)

StopableWSGIServer is a WSGIServer which run in a separated thread. This allow to use tools like casperjs or selenium.

Server instance have an `application_url` attribute formated with the server host and port.

classmethod create(*application*, ***kwargs*)

Start a server to serve `application`. Return a server instance.

run()

Run the server

shutdown()

Shutdown the server

wait(*retries=30*)

Wait until the server is started

wrapper(*environ*, *start_response*)

Wrap the wsgi application to override some path:

`/__application__`: allow to ping the server.

`/__file__?__file__={path}`: serve the file found at `path`

webtest.http.**check_server**(*host*, *port*, *path_info='/'*, *timeout=3*, *retries=30*)

Perform a request until the server reply

5.1.4 WSGI Debug application

`webtest.debugapp.debug_app` is a faker WSGI app to help to test *webtest*.

Examples of use :

```
>>> import webtest
>>> from webtest.debugapp import debug_app
>>> app = webtest.TestApp(debug_app)
>>> res = app.post('/', params='foobar')
>>> print(res.body)
CONTENT_LENGTH: 6
CONTENT_TYPE: application/x-www-form-urlencoded
HTTP_HOST: localhost:80
...
```

```
wsgi.url_scheme: 'http'
wsgi.version: (1, 0)
-- Body ----------
foobar
```

Here, you can see, `foobar` in *body* when you pass `foobar` in `app.post` `params` argument.

You can also define the status of response :

```
>>> res = app.post('/?status=302', params='foobar')
>>> print(res.status)
302 Found
```

5.1.5 Framework Hooks

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

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

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

5.2 `webtest` API

Routines for testing WSGI applications.

5.2.1 `webtest.app.TestApp`

class `webtest.app.`**`TestApp`**(*app*, *extra_environ=None*, *relative_to=None*, *use_unicode=True*, *cookiejar=None*)

Wraps a WSGI application in a more convenient interface for testing. It uses extended version of `webob.BaseRequest` and `webob.Response`.

Parameters

- **app** (*WSGI application*) – May be an WSGI application or Paste Deploy app, like `'config:filename.ini#test'`. New in version 2.0. It can also be an actual full URL to an http server and webtest will proxy requests with *wsgiproxy*.
- **extra_environ** (*dict*) – A dictionary of values that should go into the environment for each request. These can provide a communication channel with the application.
- **relative_to** (*string*) – A directory used for file uploads are calculated relative to this. Also `config:` URIs that aren't absolute.
- **cookiejar** (*CookieJar instance*) – `cookielib.CookieJar` alike API that keeps cookies across requets.

`cookies`

A convenient shortcut for a dict of all cookies in `cookiejar`.

RequestClass
alias of `TestRequest`

delete(*url*, *params=u''*, *headers=None*, *extra_environ=None*, *status=None*, *expect_errors=False*, *content_type=None*)
Do a DELETE request. Similar to `get()`.

Returns `webtest.TestResponse` instance.

delete_json(*url*, *params=<NoDefault>*, ***kw*)
Do a DELETE request. Very like the `delete` method.

`params` are dumped to json and put in the body of the request. Content-Type is set to `application/json`.

Returns a `webtest.TestResponse` object.

do_request(*req*, *status*, *expect_errors*)
Executes the given webob Request (`req`), with the expected `status`. Generally `get()` and `post()` are used instead.

To use this:

```
req = webtest.TestRequest.blank('url', ...args...)
resp = app.do_request(req)
```

Note: You can pass any keyword arguments to `TestRequest.blank()`, which will be set on the request. These can be arguments like `content_type`, `accept`, etc.

---

encode_multipart(*params*, *files*)
Encodes a set of parameters (typically a name/value list) and a set of files (a list of (name, filename, file_body)) into a typical POST body, returning the (content_type, body).

get(*url*, *params=None*, *headers=None*, *extra_environ=None*, *status=None*, *expect_errors=False*)
Do a GET request given the url path.

Parameters

- **params** – A query string, or a dictionary that will be encoded into a query string. You may also include a URL query string on the `url`.
- **headers** (*dictionary*) – Extra headers to send.
- **extra_environ** (*dictionary*) – Environmental variables that should be added to the request.
- **status** (*integer or string*) – The HTTP status code you expect in response (if not 200 or 3xx). You can also use a wildcard, like `'3*'` or `'*'`.
- **expect_errors** (*boolean*) – If this is False, then if anything is written to environ `wsgi.errors` it will be an error. If it is True, then non-200/3xx responses are also okay.

Returns `webtest.TestResponse` instance.

head(*url*, *headers=None*, *extra_environ=None*, *status=None*, *expect_errors=False*)
Do a HEAD request. Similar to `get()`.

Returns `webtest.TestResponse` instance.

options(*url*, *headers=None*, *extra_environ=None*, *status=None*, *expect_errors=False*)
Do a OPTIONS request. Similar to `get()`.

Returns `webtest.TestResponse` instance.

patch(*url*, *params=u''*, *headers=None*, *extra_environ=None*, *status=None*, *upload_files=None*, *expect_errors=False*, *content_type=None*)
Do a PATCH request. Similar to `post()`.

Returns `webtest.TestResponse` instance.

patch_json(*url*, *params=<NoDefault>*, ***kw*)
Do a PATCH request. Very like the `patch` method.

`params` are dumped to json and put in the body of the request. Content-Type is set to `application/json`.

Returns a `webtest.TestResponse` object.

post(*url*, *params=u''*, *headers=None*, *extra_environ=None*, *status=None*, *upload_files=None*, *expect_errors=False*, *content_type=None*)
Do a POST request. Similar to `get()`.

Parameters

- **params** – Are put in the body of the request. If params is a iterator it will be urlencoded, if it is string it will not be encoded, but placed in the body directly.

  Can be a collections.OrderedDict with `webtest.forms.Upload` fields included:

  app.post('/myurl', collections.OrderedDict([('textfield1', 'value1'), ('uploadfield', webapp.Upload('filename.txt', 'contents'), ('textfield2', 'value2')])))

- **upload_files** (*list*) – It should be a list of `(fieldname, filename, file_content)`. You can also use just `(fieldname, filename)` and the file contents will be read from disk.
- **content_type** (*string*) – HTTP content type, for example *application/json*.

Returns `webtest.TestResponse` instance.

post_json(*url*, *params=<NoDefault>*, ***kw*)
Do a POST request. Very like the `post` method.

`params` are dumped to json and put in the body of the request. Content-Type is set to `application/json`.

Returns a `webtest.TestResponse` object.

put(*url*, *params=u''*, *headers=None*, *extra_environ=None*, *status=None*, *upload_files=None*, *expect_errors=False*, *content_type=None*)
Do a PUT request. Similar to `post()`.

Returns `webtest.TestResponse` instance.

put_json(*url*, *params=<NoDefault>*, ***kw*)
Do a PUT request. Very like the `put` method.

`params` are dumped to json and put in the body of the request. Content-Type is set to `application/json`.

Returns a `webtest.TestResponse` object.

request(*url_or_req*, *status=None*, *expect_errors=False*, ***req_params*)
Creates and executes a request. You may either pass in an instantiated `TestRequest` object, or you may pass in a URL and keyword arguments to be passed to `TestRequest.blank()`.

You can use this to run a request without the intermediary functioning of `TestApp.get()` etc. For instance, to test a WebDAV method:

```
resp = app.request('/new-col', method='MKCOL')
```

Note that the request won't have a body unless you specify it, like:

```
resp = app.request('/test.txt', method='PUT', body='test')
```

You can use `webtest.TestRequest`:

```
req = webtest.TestRequest.blank('/url/', method='GET')
resp = app.do_request(req)
```

reset()
Resets the state of the application; currently just clears saved cookies.

5.2.2 webtest.app.TestRequest

class webtest.app.**TestRequest**(*environ*, *charset=None*, *unicode_errors=None*, *decode_param_names=None*, ***kw*)
Bases: webob.request.BaseRequest

A subclass of webob.Requset

ResponseClass
alias of TestResponse

5.2.3 webtest.response.TestResponse

class webtest.response.**TestResponse**(*body=None*, *status=None*, *headerlist=None*, *app_iter=None*, *content_type=None*, *conditional_response=None*, ***kw*)
Bases: webob.response.Response

Instances of this class are returned by TestApp

click(*description=None*, *linkid=None*, *href=None*, *index=None*, *verbose=False*, *extra_environ=None*)
Click the link as described. Each of `description`, `linkid`, and `url` are *patterns*, meaning that they are either strings (regular expressions), compiled regular expressions (objects with a `search` method), or callables returning true or false.

All the given patterns are ANDed together:

- `description` is a pattern that matches the contents of the anchor (HTML and all – everything between `<a...>` and `</a>`)
- `linkid` is a pattern that matches the `id` attribute of the anchor. It will receive the empty string if no id is given.
- `href` is a pattern that matches the `href` of the anchor; the literal content of that attribute, not the fully qualified attribute.

If more than one link matches, then the `index` link is followed. If `index` is not given and more than one link matches, or if no link matches, then `IndexError` will be raised.

If you give `verbose` then messages will be printed about each link, and why it does or doesn't match. If you use `app.click(verbose=True)` you'll see a list of all the links.

You can use multiple criteria to essentially assert multiple aspects about the link, e.g., where the link's destination is.

clickbutton(*description=None*, *buttonid=None*, *href=None*, *index=None*, *verbose=False*)
Like `click()`, except looks for link-like buttons. This kind of button should look like `<button onclick="...location.href='url'...">`.

follow(***kw*)
If this request is a redirect, follow that redirect. It is an error if this is not a redirect response. Any keyword arguments are passed to `webtest.TestApp`. Returns another response object.

form
If there is only one form on the page, return it as a `Form` object; raise a TypeError is there are no form or multiple forms.

forms
Returns a dictionnary containing all the forms in the pages as `Form` objects. Indexes are both in order (from zero) and by form id (if the form is given an id).

See *Form handling* for more info on form objects.

goto(*href*, *method='get'*, ***args*)
Go to the (potentially relative) link `href`, using the given method (`'get'` or `'post'`) and any extra arguments you want to pass to the `webtest.TestApp.get()` or `webtest.TestApp.post()` methods.

All hostnames and schemes will be ignored.

html
Returns the response as a BeautifulSoup object.

Only works with HTML responses; other content-types raise AttributeError.

json
Return the response as a JSON response. You must have simplejson installed to use this, or be using a Python version with the json module.

The content type must be application/json to use this.

lxml
Returns the response as an lxml object. You must have lxml installed to use this.

If this is an HTML response and you have lxml 2.x installed, then an `lxml.html.HTML` object will be returned; if you have an earlier version of lxml then a `lxml.HTML` object will be returned.

mustcontain(**strings*, *no=[]*)
Assert that the response contains all of the strings passed in as arguments.

Equivalent to:

```
assert string in res
```

Can take a *no* keyword argument that can be a string or a list of strings which must not be present in the response.

normal_body
Return the whitespace-normalized body

pyquery
Returns the response as a PyQuery object.

Only works with HTML and XML responses; other content-types raise AttributeError.

showbrowser()
Show this response in a browser window (for debugging purposes, when it's hard to read the HTML).

unicode_normal_body
Return the whitespace-normalized body, as unicode

xml
Returns the response as an ElementTree object.

Only works with XML responses; other content-types raise AttributeError

5.2.4 webtest.forms

Helpers to fill and submit forms.

class webtest.forms.**Checkbox**(**args*, ***attrs*)
Bases: webtest.forms.Field

Field representing <input type="checkbox">

checked
Returns True if checkbox is checked.

force_value(*value*)
Like setting a value, except forces it (even for, say, hidden fields).

class webtest.forms.**Field**(*form*, *tag*, *name*, *pos*, *value=None*, *id=None*, ***attrs*)
Bases: object

Base class for all Field objects.

classes
Dictionary of field types (select, radio, etc)

value
Set/get value of the field.

force_value(*value*)
Like setting a value, except forces it (even for, say, hidden fields).

class webtest.forms.**File**(*form*, *tag*, *name*, *pos*, *value=None*, *id=None*, ***attrs*)
Bases: webtest.forms.Field

Field representing <input type="file">

force_value(*value*)
Like setting a value, except forces it (even for, say, hidden fields).

class webtest.forms.**Form**(*response*, *text*)
Bases: object

This object represents a form that has been found in a page.

Parameters

- **response** – *webob.response.TestResponse* instance
- **text** – Unparsed html of the form

text
the full HTML of the form.

action
the relative URI of the action.

method
the HTTP method (e.g., 'GET').

id
the id, or None if not given.

enctype
encoding of the form submission

fields
a dictionary of fields, each value is a list of fields by that name. `<input type="radio">` and `<select>` are both represented as single fields with multiple options.

field_order
Ordered list of field names as found in the html.

FieldClass
alias of `Field`

get(*name*, *index=None*, *default=<NoDefault>*)
Get the named/indexed field object, or `default` if no field is found. Throws an AssertionError if no field is found and no `default` was given.

lint()
Check that the html is valid:

- each field must have an id
- each field must have a label

select(*name*, *value*, *index=None*)
Like `.set()`, except also confirms the target is a `<select>`.

set(*name*, *value*, *index=None*)
Set the given name, using `index` to disambiguate.

submit(*name=None*, *index=None*, ***args*)
Submits the form. If `name` is given, then also select that button (using `index` to disambiguate)‘‘.

Any extra keyword arguments are passed to the `webtest.TestResponse.get()` or `webtest.TestResponse.post()` method.

Returns a `webtest.TestResponse` object.

submit_fields(*name=None*, *index=None*)
Return a list of `[(name, value), ...]` for the current state of the form.

Parameters

- **name** – Same as for `submit()`
- **index** – Same as for `submit()`

upload_fields()
Return a list of file field tuples of the form:

```
(field name, file name)
```

or:

```
(field name, file name, file contents).
```

class webtest.forms.**Hidden**(*form*, *tag*, *name*, *pos*, *value=None*, *id=None*, ***attrs*)
Bases: `webtest.forms.Text`

Field representing `<input type="hidden">`

force_value(*value*)
Like setting a value, except forces it (even for, say, hidden fields).

class webtest.forms.**MultipleSelect**(**args*, ***attrs*)
Bases: webtest.forms.Field

Field representing <select multiple="multiple">

force_value(*values*)
Like setting a value, except forces it (even for, say, hidden fields).

class webtest.forms.**Radio**(**args*, ***attrs*)
Bases: webtest.forms.Select

Field representing <input type="radio">

force_value(*value*)
Like setting a value, except forces it (even for, say, hidden fields).

class webtest.forms.**Select**(**args*, ***attrs*)
Bases: webtest.forms.Field

Field representing <select /> form element.

force_value(*value*)
Like setting a value, except forces it (even for, say, hidden fields).

class webtest.forms.**Submit**(*form*, *tag*, *name*, *pos*, *value=None*, *id=None*, ***attrs*)
Bases: webtest.forms.Field

Field representing <input type="submit"> and <button>

force_value(*value*)
Like setting a value, except forces it (even for, say, hidden fields).

class webtest.forms.**Text**(*form*, *tag*, *name*, *pos*, *value=None*, *id=None*, ***attrs*)
Bases: webtest.forms.Field

Field representing <input type="text">

force_value(*value*)
Like setting a value, except forces it (even for, say, hidden fields).

class webtest.forms.**Textarea**(*form*, *tag*, *name*, *pos*, *value=None*, *id=None*, ***attrs*)
Bases: webtest.forms.Text

Field representing <textarea>

force_value(*value*)
Like setting a value, except forces it (even for, say, hidden fields).

class webtest.forms.**Upload**(*filename*, *content=None*)
Bases: object

A file to upload:

```
>>> Upload('filename.txt', 'data')
<Upload "filename.txt">
>>> Upload("README.txt")
<Upload "README.txt">
```

Parameters

- **filename** – Name of the file to upload.

- **content** – Contents of the file.

5.2.5 webtest.http

This module contains some helpers to deal with the real http world.

class webtest.http.**StopableWSGIServer**(*application*, **args*, ***kwargs*)
Bases: waitress.server.WSGIServer

StopableWSGIServer is a WSGIServer which run in a separated thread. This allow to use tools like casperjs or selenium.

Server instance have an `application_url` attribute formated with the server host and port.

classmethod create(*application*, ***kwargs*)
Start a server to serve `application`. Return a server instance.

run()
Run the server

shutdown()
Shutdown the server

wait(*retries=30*)
Wait until the server is started

wrapper(*environ*, *start_response*)
Wrap the wsgi application to override some path:

`/__application__`: allow to ping the server.

`/__file__?__file__={path}`: serve the file found at `path`

webtest.http.**check_server**(*host*, *port*, *path_info='/'*, *timeout=3*, *retries=30*)
Perform a request until the server reply

5.2.6 webtest.lint

Middleware to check for obedience to the WSGI specification.

Some of the things this checks:

- Signature of the application and start_response (including that keyword arguments are not used).
- Environment checks:
  - Environment is a dictionary (and not a subclass).
  - That all the required keys are in the environment: REQUEST_METHOD, SERVER_NAME, SERVER_PORT, wsgi.version, wsgi.input, wsgi.errors, wsgi.multithread, wsgi.multiprocess, wsgi.run_once
  - That HTTP_CONTENT_TYPE and HTTP_CONTENT_LENGTH are not in the environment (these headers should appear as CONTENT_LENGTH and CONTENT_TYPE).
  - Warns if QUERY_STRING is missing, as the cgi module acts unpredictably in that case.
  - That CGI-style variables (that don't contain a .) have (non-unicode) string values
  - That wsgi.version is a tuple
  - That wsgi.url_scheme is 'http' or 'https' (@@: is this too restrictive?)

  - Warns if the REQUEST_METHOD is not known (@@: probably too restrictive).
  - That SCRIPT_NAME and PATH_INFO are empty or start with /
  - That at least one of SCRIPT_NAME or PATH_INFO are set.
  - That CONTENT_LENGTH is a positive integer.
  - That SCRIPT_NAME is not '/' (it should be '', and PATH_INFO should be '/').
  - That wsgi.input has the methods read, readline, readlines, and __iter__
  - That wsgi.errors has the methods flush, write, writelines
- The status is a string, contains a space, starts with an integer, and that integer is in range (> 100).
- That the headers is a list (not a subclass, not another kind of sequence).
- That the items of the headers are tuples of strings.
- That there is no 'status' header (that is used in CGI, but not in WSGI).
- That the headers don't contain newlines or colons, end in _ or -, or contain characters codes below 037.
- That Content-Type is given if there is content (CGI often has a default content type, but WSGI does not).
- That no Content-Type is given when there is no content (@@: is this too restrictive?)
- That the exc_info argument to start_response is a tuple or None.
- That all calls to the writer are with strings, and no other methods on the writer are accessed.
- That wsgi.input is used properly:
  - .read() is called with zero or one argument
  - That it returns a string
  - That readline, readlines, and __iter__ return strings
  - That .close() is not called
  - No other methods are provided
- That wsgi.errors is used properly:
  - .write() and .writelines() is called with a string, except with python3
  - That .close() is not called, and no other methods are provided.
- The response iterator:
  - That it is not a string (it should be a list of a single string; a string will work, but perform horribly).
  - That .next() returns a string
  - That the iterator is not iterated over until start_response has been called (that can signal either a server or application error).
  - That .close() is called (doesn't raise exception, only prints to sys.stderr, because we only know it isn't called when the object is garbage collected).

`webtest.lint.`**`middleware`**(*application*, *global_conf=None*)

When applied between a WSGI server and a WSGI application, this middleware will check for WSGI compliancy on a number of levels. This middleware does not modify the request or response in any way, but will throw an AssertionError if anything seems off (except for a failure to close the application iterator, which will be printed to stderr – there's no way to throw an exception at that point).

5.2.7 webtest.debugapp

class webtest.debugapp.**DebugApp**(*form=None*, *show_form=False*)
Bases: object

The WSGI application used for testing

webtest.debugapp.**make_debug_app**(*global_conf*, ***local_conf*)
An application that displays the request environment, and does nothing else (useful for debugging and test purposes).

5.3 Contribute to webtest project

5.3.1 Getting started

Get your working copy :

```
$ git clone https://github.com/Pylons/webtest.git
$ cd webtest
$ virtualenv .
$ . bin/activate
$ python setup.py dev
```

Now, you can hack.

5.3.2 Execute tests

```
$ bin/nosetests
Doctest: forms.txt ... ok
Doctest: index.txt ... ok

...

test_url_class (tests.test_testing.TestTesting) ... ok
tests.test_testing.test_print_unicode ... °C
ok

Name                Stmts   Miss  Cover   Missing
-------------------------------------------------
webtest                18      0   100%
webtest.app           603     92    85%   48, 61-62, 94, 98, 212-221, 264-265, 268-272, 347, 379-386,
webtest.compat         50     11    78%   28-34, 55-56, 61-62
webtest.debugapp       58      0   100%
webtest.ext            80      0   100%
webtest.forms         324     23    93%   23, 49, 58, 61, 92, 116, 177, 205, 411, 478, 482-486, 491-49
webtest.http           78      0   100%
webtest.lint          215     45    79%   135, 176, 214-216, 219-224, 227-231, 234, 243-244, 247, 250-
webtest.sel           479    318    34%   38-39, 45-46, 64-78, 88-108, 120, 126, 151-153, 156-158, 164
webtest.utils          99     11    89%   19-20, 23, 26, 32, 38, 100, 109, 152-154
-------------------------------------------------
TOTAL                2004    500    75%
----------------------------------------------------------------------
Ran 70 tests in 14.940s
```

5.3.3 Use tox to test many Python versions

Tox installation :

```
$ pip install tox
$ tox
```

Launch tests with *tox* :

```
$ bin/tox
py26: commands succeeded
py27: commands succeeded
py32: commands succeeded
py33: commands succeeded
```

To execute test on all python versions, you need to have `python2.6`, `python2.7`, `python3.2` and `python3.3` in your `PATH`.

5.3.4 Generate documentation

```
$ pip install Sphinx
$ cd docs
$ make html
../bin/sphinx-build -b html -d _build/doctrees   . _build/html
Running Sphinx v1.1.3
loading pickled environment... done

...

build succeeded, 3 warnings.

Build finished. The HTML pages are in _build/html.
```

5.3.5 Tips

You can use *WSGI Debug application* object to test *webtest*.

5.4 News

5.4.1 2.0dev

- drop zc.buildout usage for development, now using only virtualenv [Domen Kožar]
- Backward incompatibility : Removed the `anchor` argument of `TestResponse.click()` and the `button` argument of `TestResponse.clickbutton()`. It is for the greater good. [madjar]
- Rewrote API documentation [Domen Kožar]
- Added *wsgiproxy* support to do HTTP request to an URL [gawel]
- Use BeautifulSoup4 to parse forms [gawel]
- Added *TestApp.patch_json* [gawel]

- Implement *TestApp.cookiejar* support and kindof keep *TestApp.cookies* functionality. *TestApp.cookies* should be treated as read-only. [Domen Kožar]
- Split Selenium integration into separate package webtest-selenium [gawel]
- Split casperjs integration into separate package webtest-casperjs [gawel]
- Test coverage improvements [harobed, cdevienne, arthru, Domen Kožar, gawel]
- Fully implement decoding of HTML entities
- Fix tox configuration

5.4.2 1.4.2

- fix tests error due to CLRF in a tarball

5.4.3 1.4.1

- add travis-ci
- migrate repository to https://github.com/Pylons/webtest
- Fix a typo in apps.py: selectedIndicies
- Preserve field order during parsing (support for deform and such)
- allow equals sign in the cookie by spliting name-value-string pairs on the first '=' sign as per http://tools.ietf.org/html/rfc6265#section-5.2
- fix an error when you use AssertionError(response) with unicode chars in response

5.4.4 1.4.0

- added webtest.ext - allow to use casperjs

5.4.5 1.3.6

- fix #42 Check uppercase method.
- fix #36 Radio can use forced value.
- fix #24 Include test fixtures.
- fix bug when trying to print a response which contain some unicode chars

5.4.6 1.3.5

- fix #39 Add PATCH to acceptable methods.

5.4.7 1.3.4

- fix #33 Remove CaptureStdout. Do nothing and break pdb
- use OrderedDict to store fields in form. See #31
- fix #38 Allow to post falsey values.
- fix #37 Allow Content-Length: 0 without Content-Type
- fix #30 bad link to pyquery documentation
- Never catch NameError during iteration

5.4.8 1.3.3

- added `post_json`, `put_json`, `delete_json`
- fix #25 params dictionary of webtest.AppTest.post() does not support unicode values

5.4.9 1.3.2

- improve showbrowser. fixed #23
- print_stderr fail with unicode string on python2

5.4.10 1.3.1

- Added .option() #20
- Fix #21
- Full python3 compat

5.4.11 1.3

- Moved TestApp to app.py
- Added selenium testing framework. See `sel` module.

5.4.12 1.2.4

- Accept lists for `app.post(url, params=[...])`
- Allow to use url that starts with the SCRIPT_NAME found in extra_environ
- Fix #16 Default content-type is now correctly set to *application/octet-stream*
- Fix #14 and #18 Allow to use *.delete(params={})*
- Fix #12

5.4.13 1.2.3

- Fix #10, now *TestApp.extra_environ* doesn't take precedence over a WSGI environment passed in through the request.
- Removed stray print

5.4.14 1.2.2

- Revert change to cookies that would add `"` around cookie values.
- Added property `webtest.Response.pyquery()` which returns a PyQuery object.
- Set base_url on `resp.lxml`
- Include tests and docs in tarball.
- Fix sending in webob.Request (or webtest.TestRequest) objects.
- Fix handling forms with file uploads, when no file is selected.
- Added `extra_environ` argument to `webtest.TestResponse.click()`.
- Fixed/added wildcard statuses, like `status="4*"`
- Fix file upload fields in forms: allow upload field to be empty.
- Added support for single-quoted html attributes.
- *TestResponse* now has unicode support. It is turned on by default for all responses with charset information. **This is backward incompatible change** if you rely (e.g. in doctests) on parsed form fields or responses returned by *json* and *lxml* methods being encoded strings when charset header is in response. In order to switch to old behaviour pass *use_unicode=False* flag to *TestApp* constructor.

5.4.15 1.2.1

- Added method `TestApp.request()`, which can be used for sending requests with different methods (e.g., `MKCOL`). This method sends all its keyword arguments to `webtest.TestRequest.blank()` and then executes the request. The parameters are somewhat different than other methods (like `webtest.TestApp.get()`), as they match WebOb's attribute names exactly (the other methods were written before WebOb existed).
- Removed the copying of stdout to stderr during requests.
- Fix file upload fields in forms (#340) – you could upload files with `webtest.TestApp.post()`, but if you use `resp.form` file upload fields would not work (from rcs-comp.com and Matthew Desmarais).

5.4.16 1.2

- Fix form inputs; text inputs always default to the empty string, and unselected radio inputs default to nothing at all. From Daniele Paolella.
- Fix following links with fragments (these fragments should not be sent to the WSGI application). From desmaj.
- Added `force_value` to select fields, like `res.form['select'].force_value("new_value")`. This makes it possible to simulate forms that are dynamically updated. From Matthew Desmarais.
- Fixed `webtest.Response.mustcontain()` when you pass in a `no=[strings]` argument.

5.4.17 1.1

- Changed the `__str__` of responses to make them more doctest friendly:
  - All headers are displayed capitalized, like Content-Type
  - Headers are sorted alphabetically
- Changed `__repr__` to only show the body length if the complete body is not shown (for short bodies the complete body is in the repr)
- Note: **these are backward incompatible changes** if you are using doctest (you'll have to update your doctests with the new format).
- Fixed exception in the `.delete` method.
- Added a `content_type` argument to `app.post` and `app.put`, which sets the `Content-Type` of the request. This is more convenient when testing REST APIs.
- Skip links in `<script>...</script>` tags (since that's not real markup).

5.4.18 1.0.2

- Don't submit unnamed form fields.
- Checkboxes with no explicit `value` send `on` (previously they sent `checked`, which isn't what browsers send).
- Support for `<select multiple>` fields (from Matthew Desmarais)

1.0.1 —

- Fix the `TestApp` validator's InputWrapper lacking support for readline with an argument as needed by the cgi module.

5.4.19 1.0

- Keep URLs in-tact in cases such as `app.get('http://www.python.org')` (so HTTP_HOST=www.python.org, etc).
- Fix `lxml.html` import, so lxml 2.0 users can get HTML lxml objects from `resp.lxml`
- Treat `<input type="image">` like a submit button.
- Use `BaseCookie` instead of `SimpleCookie` for storing cookies (avoids quoting cookie values).
- Accept any `params` argument that has an `items` method (like MultiDict)

5.4.20 0.9

Initial release

CHAPTER

SIX

LICENSE

PYTHON MODULE INDEX