WebTest Documentation

Release 1.4.3

Ian Bicking

March 28, 2013

CONTENTS

author Ian Bicking <ianb@colorstudy.com>

maintainer Gael Pasgrimaud <gael@gawel.org>

Contents

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 Paste 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

TESTAPP

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.)

```
>>> from webtest import TestApp
>>> from webtest.debugapp import debug_app
>>> app = TestApp(debug_app)
>>> res = app.get('/form.html')
>>> res.status
'200 OK'
>>> res.form
<Form />
```

4.1 Making Requests

To make a request, use:

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

This does a request for `/path`, with any extra headers or WSGI environment keys that you indicate. This returns a response object, based on webob.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 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, fild_content)]`. File uploads use a different form submission data type to pass the structured data.

For other verbs you can use:

```
app.put(path, params, ...)
app.delete(path, ...)
```

These do PUT and DELETE requests.

4.1.1 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'))
```

4.1.2 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`.

4.2 The Response Object

The response object is based on webob.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.headers`: A dictionary-like object of the headers in the response.

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

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

`response.request`: The webob.Request 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 to `app.get` (e.g., `status`).

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

`response.mustcontain(string1, string2, ...)`: Raises an error if any of the strings are not 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 docstring for more)

`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 Submissions* 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.

CHAPTER

FIVE

FORM SUBMISSIONS

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

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

Then you fill it in fields:

```
>>> print(form.action)
/form-submit
>>> print(form.method)
POST
>>> # dict of fields
>>> form.fields.values()
[(u'name', [<Text name="name">]), (u'submit', [<Submit name="submit">])]
>>> form['name'] = 'Bob'
>>> # When names don't point to a single field:
>>> form.set('name', 'Bob', index=0)
```

Then you can submit:

```
>>> # 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
...
-- Body ----------
name=Bob&submit=Submit%21
```

Select fields can only be set to valid values (i.e., values in an `<option>`) but you can also use `form['select-field'].force_value('value')` to enter values not present in an option.

CHAPTER

SIX

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.

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

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

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

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

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.

Examples:

```
>>> from webtest import TestRequest
>>> from webtest import TestResponse
>>> res = TestResponse(content_type='text/html', body=b'''
... <html><body><div id="content">hey!</div></body>''')
>>> res.request = TestRequest.blank('/')
>>> res.html

<html><body><div id="content">hey!</div></body></html>
>>> res.html.__class__
<class 'BeautifulSoup.BeautifulSoup'>
>>> res.lxml
<Element html at ...>
>>> res.lxml.xpath('//body/div')[0].text
'hey!'
>>> res = TestResponse(content_type='application/json',
...                    body=b'{"a":1,"b":2}')
>>> res.request = TestRequest.blank('/')
>>> list(res.json.values())
[1, 2]
>>> res = TestResponse(content_type='application/xml',
...                    body=b'<xml><message>hey!</message></xml>')
>>> res.request = TestRequest.blank('/')
>>> res.xml
<Element ...>
>>> res.xml[0].tag
'message'
>>> res.xml[0].text
'hey!'
>>> res.lxml
```

```
<Element xml at ...>
>>> res.lxml[0].tag
'message'
>>> res.lxml[0].text
'hey!'
>>> res.pyquery('message')
[<message>]
>>> res.pyquery('message').text()
'hey!'
```

CHAPTER

SEVEN

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`.

CHAPTER

EIGHT

API

8.1 `webtest` – Functional Testing of Web Applications

Routines for testing WSGI applications.

Most interesting is app

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

Wraps a WSGI application in a more convenient interface for testing.

`app` may be an application, or a Paste Deploy app URI, like `'config:filename.ini#test'`.

`extra_environ` is a dictionary of values that should go into the environment for each request. These can provide a communication channel with the application.

`relative_to` is a directory, and filenames used for file uploads are calculated relative to this. Also `config:` URIs that aren't absolute.

`delete`(*url*, *params=''*, *headers=None*, *extra_environ=None*, *status=None*, *expect_errors=False*, *content_type=None*)

Do a DELETE request. Very like the `.get()` method.

Returns a `webob.Response` object.

`delete_json`(*url*, *params=<class 'webtest.app.NoDefault'>*, *headers=None*, *extra_environ=None*, *status=None*, *expect_errors=False*)

Do a DELETE request. Very like the `.get()` method. Content-Type is set to `application/json`.

Returns a `webob.Response` object.

`do_request`(*req*, *status*, *expect_errors*)

Executes the given request (`req`), with the expected `status`. Generally `.get()` and `.post()` are used instead.

To use this:

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

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*)

Get the given url (well, actually a path like `'/page.html'`).

params: A query string, or a dictionary that will be encoded into a query string. You may also include a query string on the `url`.

headers: A dictionary of extra headers to send.

extra_environ: A dictionary of environmental variables that should be added to the request.

status: The integer status code you expect (if not 200 or 3xx). If you expect a 404 response, for instance, you must give `status=404` or it will be an error. You can also give a wildcard, like `'3*'` or `'*'`.

expect_errors: If this is not true, then if anything is written to `wsgi.errors` it will be an error. If it is true, then non-200/3xx responses are also okay.

Returns a `webtest.TestResponse` object.

head (*url*, *headers=None*, *extra_environ=None*, *status=None*, *expect_errors=False*)
Do a HEAD request. Very like the `.get()` method.

Returns a `webob.Response` object.

options (*url*, *headers=None*, *extra_environ=None*, *status=None*, *expect_errors=False*)
Do a OPTIONS request. Very like the `.get()` method.

Returns a `webob.Response` object.

post (*url*, *params=''*, *headers=None*, *extra_environ=None*, *status=None*, *upload_files=None*, *expect_errors=False*, *content_type=None*)
Do a POST request. Very like the `.get()` method. `params` are put in the body of the request.

`upload_files` is for file uploads. It should be a list of `[(fieldname, filename, file_content)]`. You can also use just `[(fieldname, filename)]` and the file content will be read from disk.

For post requests params could be a collections.OrderedDict with Upload fields included in order:

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

Returns a `webob.Response` object.

post_json (*url*, *params=<class 'webtest.app.NoDefault'>*, *headers=None*, *extra_environ=None*, *status=None*, *expect_errors=False*)
Do a POST request. Very like the `.get()` method. `params` are dumps to json and put in the body of the request. Content-Type is set to `application/json`.

Returns a `webob.Response` object.

put (*url*, *params=''*, *headers=None*, *extra_environ=None*, *status=None*, *upload_files=None*, *expect_errors=False*, *content_type=None*)
Do a PUT request. Very like the `.post()` method. `params` are put in the body of the request, if params is a tuple, dictionary, list, or iterator it will be urlencoded and placed in the body as with a POST, if it is string it will not be encoded, but placed in the body directly.

Returns a `webob.Response` object.

put_json (*url*, *params=<class 'webtest.app.NoDefault'>*, *headers=None*, *extra_environ=None*, *status=None*, *expect_errors=False*)
Do a PUT request. Very like the `.post()` method. `params` are dumps to json and put in the body of the request. Content-Type is set to `application/json`.

Returns a `webob.Response` 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 `POST={args}` to set the request body to the serialized arguments, and simultaneously set the request method to `POST`

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

exception `webtest.`**AppError** (*message*, **args*)

8.1.1 Response API

Some of the return values return instances of these classes:

class `webtest.`**TestResponse** (*body=None*, *status=None*, *headerlist=None*, *app_iter=None*, *content_type=None*, *conditional_response=None*, ***kw*)
Instances of this class are return by `TestApp`

click (*description=None*, *linkid=None*, *href=None*, *anchor=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.
- `anchor` is a pattern that matches the entire anchor, with its contents.

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*, *button=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. Returns another response object.

form
Returns a single `Form` instance; it is an error if there are multiple forms on the page.

forms
A list of :class:'~webtest.Form's found on the page

forms__get()
Returns a dictionary of `Form` objects. Indexes are both in order (from zero) and by form id (if the form is given an id).

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 `app.get()` or `app.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*, ***kw*)
Assert that the response contains all of the strings passed in as arguments.

Equivalent to:

```
assert string in res
```

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

class `webtest.`**`TestRequest`**(*environ*, *charset=None*, *unicode_errors=None*, *decode_param_names=None*, ***kw*)

`ResponseClass`
alias of `TestResponse`

class `webtest.`**`Form`**(*response*, *text*)
This object represents a form that has been found in a page. This has a couple useful attributes:

`text:` the full HTML of the form.

`action:` the relative URI of the action.

`method:` the method (e.g., `'GET'`).

`id:` the id, or None if not given.

`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.

`FieldClass`
alias of `Field`

`get`(*name*, *index=None*, *default=<class 'webtest.app.NoDefault'>*)
Get the named/indexed field object, or `default` if no field is found.

`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 `.get()` or `.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.

`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.`**`Field`**(*form*, *tag*, *name*, *pos*, *value=None*, *id=None*, ***attrs*)
Field object.

class `webtest.`**`Select`**(**args*, ***attrs*)
Field representing `<select>`

class `webtest.`**`Radio`**(**args*, ***attrs*)
Field representing `<input type="radio">`

class `webtest.`**`Checkbox`**(**args*, ***attrs*)
Field representing `<input type="checkbox">`

class `webtest.`**`Text`**(*form*, *tag*, *name*, *pos*, *value=None*, *id=None*, ***attrs*)
Field representing `<input type="text">`

class `webtest.`**`Textarea`**(*form*, *tag*, *name*, *pos*, *value=None*, *id=None*, ***attrs*)
Field representing `<textarea>`

class `webtest.`**`Hidden`**(*form*, *tag*, *name*, *pos*, *value=None*, *id=None*, ***attrs*)
Field representing `<input type="hidden">`

class `webtest.`**`Submit`**(*form*, *tag*, *name*, *pos*, *value=None*, *id=None*, ***attrs*)
Field representing `<input type="submit">` and `<button>`

8.2 `webtest.ext` – Using an external process

Allow to run an external process to test your application

class `webtest.ext.`**`TestApp`**(*app=None*, *url=None*, *timeout=30000*, *extra_environ=None*, *relative_to=None*, ***kwargs*)
Run the test application in a separate thread to allow to access it via http

`close`()
Close WSGI server if needed

`webtest.ext.`**`casperjs`**(**args*, ***kwds*)
A context manager to run a test with a `webtest.ext.TestApp`

8.2.1 Using casperjs to run tests

The js part:

The python part:

8.3 `webtest.sel` – Functional Testing with Selenium

Routines for testing WSGI applications with selenium.

Most interesting is `SeleniumApp` and the `selenium()` decorator

class `webtest.sel.`**`SeleniumApp`**(*app=None*, *url=None*, *timeout=30000*, *extra_environ=None*, *relative_to=None*, ***kwargs*)
See `webtest.TestApp`

SeleniumApp only support `GET` requests

`browser`
The current `Selenium`

`close`()
Close selenium and the WSGI server if needed

class `webtest.sel.`**`Selenium`**
Selenium RC control aka `browser`

A object use to manipulate DOM nodes. This object allow to use the underlying selenium api. See Selenium api

You can use the original method name:

```
browser.fireEvent('id=#myid", 'focus')
```

Or a more pythonic name:

```
browser.fire_event('id=#myid", 'focus')
```

Both are equal to:

```
browser.execute('fireEvent', 'id=#myid', 'focus')
```

`webtest.sel.`**`selenium`**(*obj*)

A callable usable as:

- class decorator
- function decorator
- contextmanager

8.3.1 Response API

Some of the return values return instances of these classes:

class `webtest.sel.`**`TestResponse`**(*body=None*, *status=None*, *headerlist=None*, *app_iter=None*, *content_type=None*, *conditional_response=None*, ***kw*)

`doc`

Expose a `Document`

`follow`(*status=None*, ***kw*)

If this request is a redirect, follow that redirect. It is an error if this is not a redirect response. Returns another response object.

class `webtest.sel.`**`Document`**(*resp*)

The browser document. `resp.doc.myid` is egual to `resp.doc.css('#myid')`

`button`(*description=None*, *buttonid=None*, *index=None*)

Get a button

`css`(*selector*)

Get an `Element` using a css selector

`get`(*tag*, ***kwargs*)

Return an element matching `tag`, an `attribute` and an `index`. For example:

```
resp.doc.get('input', name='go') => xpath=//input[@name="go"]
resp.doc.get('li', description='Item') => xpath=//li[.="Item"]
```

`input`(*value=None*, *name=None*, *inputid=None*, *index=None*)

Get an input field

`link`(*description=None*, *linkid=None*, *href=None*, *index=None*)

Get a link

`xpath`(*path*)

Get an `Element` using xpath

class `webtest.sel.`**`Element`**(*resp*, *locator*)

A object use to manipulate DOM nodes. This object allow to use the underlying selenium api for the specified locator. See Selenium api

You can use the original method name:

```
element.fireEvent('focus')
```

Or a more pythonic name:

```
element.fire_event('focus')
```

Both are equal to:

```
browser.execute('fireEvent', element.locator, 'focus')
```

attr(*attr*)
Return the attribute value of the element

drag_and_drop(*element*)
Drag and drop to element

eval(**expr*)
Eval a javascript expression in Selenium RC. You can use the following variables:

- s: the `selenium` object
- b: the `browserbot` object
- l: the element locator string
- e: the element itself

exist()
return true is the element is present

hasClass(*name*)
True iif the class is present

html()
Return the innerHTML of the element

text()
Return the text of the element

wait(*timeout=3000*)
Wait for an element and return this element

wait_and_click(*timeout=3000*)
Wait for an element, click on it and return this element

class `webtest.sel.`**Form**(*resp*, *id*)
See `Form`

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

Returns a `webtest.browser.TestResponse` object.

8.3.2 Environment variables

Those value are used if found in environment:

- `SELENIUM_HOST`: Default to `127.0.0.1`
- `SELENIUM_PORT`: Default to `4444`
- `SELENIUM_BIND`: IP used to bind extra servers (WSGI Server/File server). Default to `127.0.0.1`

- `SELENIUM_DRIVER`: The driver used to start the browser. Usualy something in `*chrome`, `*firefox`, `*googlechrome`. Default to `*googlechrome`. You can get the full list by running:

  ```
  $ java -jar selenium-server.jar -interactive
  cmd=getNewBrowserSession
  ```

- `SELENIUM_KEEP_OPEN`: If exist then browser session are not closed so you can introspect the problem on failure.
- `SELENIUM_JAR`: If selenium is not running then this jar is used to run selenium.

8.3.3 Examples

Testing a wsgi application

Testing the jquery.ui website

CHAPTER

NINE

NEWS

9.1 1.4.3

- fully implement decoding of HTML entities
- fix tox configuration

9.2 1.4.2

- fix tests error due to CLRF in a tarball

9.3 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

9.4 1.4.0

- added webtest.ext - allow to use casperjs

9.5 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

9.6 1.3.5

- fix #39 Add PATCH to acceptable methods.

9.7 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

9.8 1.3.3

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

9.9 1.3.2

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

9.10 1.3.1

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

9.11 1.3

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

9.12 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

9.13 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

9.14 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.

9.15 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).

9.16 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.

9.17 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).

9.18 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.

9.19 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)

9.20 0.9

Initial release

CHAPTER

TEN

LICENSE