ftw.testbrowser

Release 1.24.0

Jun 19, 2017

Contents

ftw.testbrowser is a browser library for testing Plone based web sites and applications.

CHAPTER 1

Introduction

ftw.testbrowser is a browser library for testing Plone based web sites and applications (CI).

Features

The test browser supports all the basic features:

- Visit pages of the Plone site
- Access page content
- Find nodes by CSS- and XPath-Expressions or by text
- Click on links
- Fill and submit forms
- File uploading
- Make WebDAV requests

The *ftw.testbrowser* also comes with some basic Plone page objects.

ftw.testbrowser currently does not support JavaScript.

Motivation

A test browser should have a simple but powerful API (CSS expressions), it should be fast, reliable and easy to setup and use.

The existing test browsers for Plone development were not satisfactory:

- The zope.testbrowser, which is the current standard for Plone testing does not support CSS- or XPath-Selectors, it is very limiting in form filling (buttons without names are not selectable, for example) and it leads to brittle tests.
- The splinter test browser has a zope driver and various selenium based drivers. This abstraction improves the API but it is still limiting since it bases on *zope.testbrowser*.
- The robotframework is a selenium based full-stack browser which comes with an own language and requires a huge setup. The use of selenium makes it slow and brittle and a new language needs to be learned.

There are also some more browser libraries and wrappers, usually around selenium, which often requires to open a port and make actual requests. This behavior is very time consuming and should not be done unless really necessary, which is usally for visual things (making screenshots) and JavaScript testing.

How it works

The *ftw.testbrowser* uses mechanize with plone.testing configurations / patches to directly dispatch requests in Zope.

The responses are parsed in an lxml.html document, which allows us to do all the necessary things such as selecting HTML elements or filling forms.

While querying, *ftw.testbrowser* wraps all the HTML elements into node wrappers which extend the *lxml* functionality with things such as using *CSS* selectors directly, clicking on links or filling forms based on labels.

CHAPTER 2

Quickstart

Add *ftw.testbrowser* to your testing dependencies in your *setup.py*:

```
tests_require = [
    'ftw.testbrowser',
    ]

setup(name='my.package',
      install_requires=['Plone'],
      tests_require=tests_require,
      extras_require=dict(tests=tests_require))
```

Write tests using the browser:

```
from ftw.testbrowser import browsing
from ftw.testbrowser.pages import factoriesmenu
from ftw.testbrowser.pages import plone
from ftw.testbrowser.pages import statusmessages
from plone.app.testing import PLONE_FUNCTIONAL_TESTING
from plone.app.testing import SITE_OWNER_NAME
from unittest2 import TestCase


class TestFolders(TestCase):

    layer = PLONE_FUNCTIONAL_TESTING

    @browsing
    def test_add_folder(self, browser):
        browser.login(SITE_OWNER_NAME).open()
        factoriesmenu.add('Folder')
        browser.fill({'Title': 'The Folder'}).submit()

        statusmessages.assert_no_error_messages()
        self.assertEquals('folder_listing', plone.view())
        self.assertEquals('The Folder', plone.first_heading())
```

CHAPTER 3

User documentation

Setup

For using the test browser, just decorate your test methods with the *@browsing* decorator.

```
from ftw.testbrowser import browsing
from unittest2 import TestCase
from plone.app.testing import PLONE_FUNCTIONAL_TESTING


class TestMyView(TestCase):

    layer = PLONE_FUNCTIONAL_TESTING

    @browsing
    def test_view_displays_things(self, browser):
        browser.visit(view='my_view')
```

Warning: Make sure that you use a functional testing layer!

See also:

`ftw.testbrowser.browsing()`

By default there is only one, global browser, but it is also possible to instantiate a new browser and to set it up manually:

```
from ftw.testbrowser.core import Browser

browser = Browser()
app = zope_app

with browser(app):
    browser.open()
```

Warning: Page objects and forms usually use the global browser. Creating a new browser manually will not set it as global browser and page objects / forms will not be able to access it!

Choosing the default driver

The default driver is chosen automatically, depending on whether the browser is set up with a zope app (=> `LIB_MECHANIZE`) or not (=> `LIB_REQUESTS`). The default driver can be changed on the browser instance, overriding the automatic driver selection:

```
from ftw.testbrowser.core import Browser
from ftw.testbrowser.core import LIB_MECHANIZE
from ftw.testbrowser.core import LIB_REQUESTS
from ftw.testbrowser.core import LIB_TRAVERSAL

browser = Browser()
# always use mechanize:
browser.default_driver = LIB_MECHANIZE

# or always use requests:
```

```
browser.default_driver = LIB_REQUESTS

# or use traversal in the same transactions with same connection:
browser.default_driver = LIB_TRAVERSAL
```

When using the testbrowser in a `plone.testing` layer, the driver can be chosen by using a standard `plone.testing` fixture:

```
from ftw.testbrowser import MECHANIZE_BROWSER_FIXTURE
from ftw.testbrowser import REQUESTS_BROWSER_FIXTURE
from ftw.testbrowser import TRAVERSAL_BROWSER_FIXTURE
from plone.app.testing import PLONE_FIXTURE
from plone.app.testing import FunctionalTesting


MY_FUNCTIONAL_TESTING_WITH_MECHANIZE = FunctionalTesting(
    bases=(PLONE_FIXTURE,
           MECHANIZE_BROWSER_FIXTURE),
    name='functional:mechanize')

MY_FUNCTIONAL_TESTING_WITH_REQUESTS = FunctionalTesting(
    bases=(PLONE_FIXTURE,
           REQUESTS_BROWSER_FIXTURE),
    name='functional:requests')

MY_FUNCTIONAL_TESTING_WITH_TRAVERSAL = FunctionalTesting(
    bases=(PLONE_FIXTURE,
           TRAVERSAL_BROWSER_FIXTURE),
    name='functional:traversal')
```

Visit pages

For visiting a page, use the *visit* or *open* method on the browser (those methods do the same).

Visiting the Plone site root:

```
browser.open()
print browser.url
```

See also:

`ftw.testbrowser.core.Browser.url()`

Visiting a full url:

```
browser.open('http://nohost/plone/sitemap')
```

Visiting an object:

```
folder = portal.get('the-folder')
browser.visit(folder)
```

Visit a view on an object:

```
folder = portal.get('the-folder')
browser.visit(folder, view='folder_contents')
```

The *open* method can also be used to make POST request:

```
browser.open('http://nohost/plone/login_form',
             {'__ac_name': TEST_USER_NAME,
              '__ac_password': TEST_USER_PASSWORD,
              'form.submitted': 1})
```

See also:

`ftw.testbrowser.core.Browser.open()`

Logging in

The *login* method sets the *Authorization* request header.

Login with the *plone.app.testing* default test user (*TEST_USER_NAME*):

```
browser.login().open()
```

Logging in with another user:

```
browser.login(username='john.doe', password='secret')
```

Logout and login a different user:

```
browser.login(username='john.doe', password='secret').open()
browser.reset()
browser.login().open()
```

See also:

`ftw.testbrowser.core.Browser.login()`, `ftw.testbrowser.core.Browser.reset()`

Finding elements

Elements can be found using CSS-Selectors (*css* method) or using XPath-Expressions (*xpath* method). A result set (*Nodes*) of all matches is returned.

See also:

`ftw.testbrowser.nodes.Nodes()`

CSS:

```
browser.open()
heading = browser.css('.documentFirstHeading').first
self.assertEquals('Plone Site', heading.normalized_text())
```

See also:

`ftw.testbrowser.core.Browser.css()`, `ftw.testbrowser.nodes.NodeWrapper.normalized_text()`

XPath:

```
browser.open()
heading = browser.xpath('h1').first
self.assertEquals('Plone Site', heading.normalized_text())
```

See also:

`ftw.testbrowser.core.Browser.xpath()`

Finding elements by text:

```
browser.open()
browser.find('Sitemap').click()
```

The *find* method will look for theese elements (in this order):

- a link with this text (normalized, including subelements' texts)
- a field which has a label with this text
- a button which has a label with this text

See also:

`ftw.testbrowser.core.Browser.find()`

Matching text content

In HTML, most elements can contain direct text but the elements can also contain sub-elements which also have text.

When having this HTML:

```
<a id="link">
    This is
    <b>a link
</a>
```

We can get only direct text of the link:

```
>>> browser.css('#link').first.text
'\n        This is\n        '
```

or the text recursively:

```
>>> browser.css('#link').first.text_content()
'\n        This is\n        a link\n    '
```

See also:

`ftw.testbrowser.nodes.NodeWrapper.text_content()`

or the normalized recursive text:

```
>>> browser.css('#link').first.normalized_text()
'This is a link'
```

See also:

`ftw.testbrowser.nodes.NodeWrapper.normalized_text()`

Functions such as *find* usually use the *normalized_text*.

See also:

`ftw.testbrowser.core.Browser.find()`

Get the page contents / json data

The page content of the currently loaded page is always available on the browser:

```
browser.open()
print browser.contents
```

See also:

`ftw.testbrowser.core.Browser.contents()`

If the result is a JSON string, you can access the JSON data (converted to python data structure already) with the *json* property:

```
browser.open(view='a-json-view')
print browser.json
```

See also:

`ftw.testbrowser.core.Browser.json()`

Filling and submitting forms

The browser's *fill* method helps to easily fill forms by label text without knowing the structure and details of the form:

```
browser.visit(view='login_form')
browser.fill({'Login Name': TEST_USER_NAME,
              'Password': TEST_USER_PASSWORD}).submit()
```

The *fill* method returns the browser instance which can be submitted with *submit*. The keys of the dict with the form data can be either field labels (<*label*> text) or the name of the field. Only one form can be filled at a time.

File uploading

For uploading a file you need to pass at least the file data (string or stream) and the filename to the *fill* method, optionally you can also declare a mime type.

There are two syntaxes which can be used.

Tuple syntax:

```
browser.fill({'File': ('Raw file data', 'file.txt', 'text/plain')})
```

Stream syntax

```
file_ = StringIO('Raw file data')
file_.filename = 'file.txt'
file_.content_type = 'text/plain'

browser.fill({'File': file_})
```

You can also pass in filesystem files directly, but you need to make sure that the file stream is opened untill the form is submitted.

```
with open('myfile.pdf') as file_:
    browser.fill({'File': file_}).submit()
```

See also:

`ftw.testbrowser.core.Browser.fill()`, `ftw.testbrowser.form.Form.submit()`, `ftw.testbrowser.form.Form.save()`

Tables

Tables are difficult to test without the right tools. For making the tests easy and readable, the table components provide helpers especially for easily extracting a table in a readable form.

For testing the content of this table:

```
<table id="shopping-cart">
    <thead>
        <tr>
            <th>Product</th>
            <th>Price</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td>Socks</td>
            <td>12.90</td>
        </tr>
        <tr>
            <td>Pants</td>
            <td>35.00</td>
        </tr>
    </tbody>
    <tfoot>
        <tr>
            <td>TOTAL:</td>
            <td>47.90</td>
        </tr>
    </tfoot>
</table>
```

You could use the `lists` method:

```
self.assertEquals(
    [['Product', 'Price'],
     ['Socks', '12.90'],
     ['Pants', '35.00'],
     ['TOTAL:', '47.90']],
    browser.css('#shopping-cart').first.lists())
```

See also:

`ftw.testbrowser.table.Table.lists()`

or the `dicts` method:

```
self.assertEquals(
    [{'Product': 'Socks',
      'Price': '12.90'},
     {'Product': 'Pants',
      'Price': '35.00'},
     {'Product': 'TOTAL:',
      'Price': '47.90'}],
    browser.css('#shopping-cart').first.dicts())
```

See also:

`ftw.testbrowser.table.Table.dicts()`

See the tables API for more details.

See also:

`ftw.testbrowser.table.Table()`, `ftw.testbrowser.table.TableRow()`, `ftw.testbrowser.table.TableCell()`

Page objects

ftw.testbrowser ships some basic page objects for Plone. Page objects represent a page or a part of a page and provide an API to this part. This allows us to write simpler and more expressive tests and makes the tests less brittle.

Read the post by Martin Fowler for better explenation about what page objects are.

You can and should write your own page objects for your views and pages.

See the API documentation for the page objects included in *ftw.testbrowser*:

- The **plone** page object provides general information about this page, such as if the user is logged in or the view / portal type of the page.
- The **factoriesmenu** page object helps to add new content through the browser or to test the addable types.
- The **statusmessages** page object helps to assert the current status messages.
- The **dexterity** page object provides helpers related to dexterity
- The **z3cform** page object provides helpers related to z3cforms, e.g. for asserting validation errors in the form.

See also:

`ftw.testbrowser.pages`

XML Support

When the response mimetype is `text/xml` or `application/xml`, the response body is parsed as XML instead of HTML.

This can lead to problems when having XML-Documents with a default namespace, because lxml only supports XPath 1, which does not support default namespaces.

You can either solve the problem yourself by parsing the `browser.contents` or you may switch back to HTML parsing. HTML parsing will modify your document though, it will insert a `html` node for example.

Re-parsing with another parser:

```
browser.webdav(view='something.xml')   # XML document
browser.parse_as_html()                # HTML document
browser.parse_as_xml()                 # XML document
```

See also:

`ftw.testbrowser.core.Browser.parse_as_html`

See also:

`ftw.testbrowser.core.Browser.parse_as_xml`

See also:

`ftw.testbrowser.core.Browser.parse`

WebDAV requests

ftw.testbrowser supports doing WebDAV requests, although it requires a ZServer to be running because of limitations in mechanize.

Use a testing layer which bases on `plone.app.testing.PLONE_ZSERVER`:

```
from plone.app.testing import FunctionalTesting
from plone.app.testing import PLONE_FIXTURE
from plone.app.testing import PLONE_ZSERVER
from plone.app.testing import PloneSandboxLayer


class MyPackageLayer(PloneSandboxLayer):

    defaultBases = (PLONE_FIXTURE, )

MY_PACKAGE_FIXTURE = MyPackageLayer()
MY_PACKAGE_ZSERVER_TESTING = FunctionalTesting(
    bases=(MY_PACKAGE_FIXTURE,
           PLONE_ZSERVER),
    name='my.package:functional:zserver')
```

Then use the `webdav` method for making requests in the test:

```
from ftw.testbrowser import browsing
from my.package.testing import MY_PACKAGE_ZSERVER_TESTING
from unittest2 import TestCase


class TestWebdav(TestCase):

    layer = MY_PACKAGE_ZSERVER_TESTING

    @browsing
    def test_DAV_option(self, browser):
        browser.webdav('OPTIONS')
        self.assertEquals('1,2', browser.response.headers.get('DAV'))
```

See also:

`ftw.testbrowser.core.Browser.webdav()`

Error handling

The testbrowser raises exceptions by default when a request was not successful. When the response has a status code of *4xx*, a `ftw.testbrowser.exceptions.HTTPClientError` is raised, when the status code is *5xx*, a `ftw.testbrowser.exceptions.HTTPServerError` is raised.

Disabling HTTP exceptions

Disable the `raise_http_errors` flag when the test browser should not raise any HTTP exceptions:

```
@browsing
def test(self, browser):
    browser.raise_http_errors = False
    browser.open(view='not-existing')
```

Expecting HTTP exceptions

Sometimes we want to make sure that the server responds with a certain bad status. For making that easy, the testbrowser provides assertion context managers:

```
@browsing
def test(self, browser):
    with browser.expect_http_error():
        browser.open(view='failing')

    with browser.expect_http_error(code=404):
        browser.open(view='not-existing')

    with browser.expect_http_error(reason='Bad Request'):
        browser.open(view='get-record-by-id')
```

Expecting unauthoirzed exceptions (Plone)

When a user is not logged in and is not authorized to access a resource, Plone will redirect the user to the login form (`require_login`). The `expect_unauthorized` context manager knows how Plone behaves and provides an easy interface so that the developer does not need to handle it.

```
@browsing
def test(self, browser):
    with browser.expect_unauthorized():
        browser.open(view='plone_control_panel')
```

Exception bubbling

Exceptions happening in views can not be catched in the browser by default. When using an internally dispatched driver such as Mechanize, the option `exception_bubbling` makes the Zope Publisher and Mechanize let the exceptions bubble up into the test method, so that it can be catched and asserted there.

```
@browsing
def test(self, browser):
    browser.exception_bubbling = True
```

```
    with self.assertRaises(ValueError) as cm:
        browser.open(view='failing')

    self.assertEquals('No valid value was submitted', str(cm.exception))
```

CHAPTER 4

API Documentation

Browser

Drivers

Drivers are responsible for making the request and responding to basic questions, such as the current URL or response headers.

RequestsDriver

MechanizeDriver

TraversalDriver

StaticDriver

Nodes and forms

Result set

Node wrappers

Node wrappers wrap the standard *lxml* elements and extend them with some useful methods so that it is nicely integrated in the *ftw.testbrowser* behavior.

Forms, fields and widgets

Tables

Page objects

Plone page object

Factories menu page object

Status messages page object

dexterity page object

z3cform page object

Exceptions

CHAPTER 5

Changelog

1.24.0 (2017-06-16)

- Log exceptions to stderr when they are not expected. [jone]
- Standardize redirect loop detection: always throw a `RedirectLoopException`. [jone]
- Add traversal request driver. [jone]

1.23.2 (2017-06-16)

- Fix *browser.context* when base_url ends with a view name. [phgross]

1.23.1 (2017-05-02)

- Fix *browser.debug* when body is a bytestring. [jone]

1.23.0 (2017-04-28)

- Introduce `browser.expect_unauthorized` context manager. [jone]

1.22.2 (2017-04-28)

- HTTPError: include code and reason in exception. [jone]
- Docs: Fix wrong expect_http_error argument names. [jone]

1.22.1 (2017-04-28)

- Docs: swith to RTD, update URLs. [jone]
- Docs: Switch to RTD Sphinx theme. [lgraf]

1.22.0 (2017-04-28)

- Forbid setting of "x-zope-handle-errors" header. [jone]
- Add an option `browser.exception_bubbling`, disabled by default. [jone]
- Mechanize: no longer disable "x-zope-handle-errors". [jone]
- Introduce `browser.expect_http_error()` context manager. [jone]
- Add an option `browser.raise_http_errors`, enabled by default. [jone]
- Raise `HTTPClientError` and `HTTPServerError` by default. [jone]
- Introduce `browser.status_reason`. [jone]
- Introduce `browser.status_code`. [jone]

1.21.0 (2017-04-19)

- Make `zope.globalrequest` support optional. [jone]
- Add testing layers for setting the default driver. [jone]
- Add `default_driver` option to the driver. [jone]
- Refactoring: introduce request drivers. [jone]

1.20.0 (2017-04-10)

- Add Support for Button tag. [tschanzt]
- No longer test with Archetypes, test only with dexterity. [jone]
- Support latest Plone 4.3.x release. [mathias.leimgruber]

1.19.3 (2016-07-25)

- Declare some previously missing test requirements. [lgraf]
- Declare previously missing dependency on zope.globalrequest (introduced in #35). [lgraf]

1.19.2 (2016-06-27)

- Preserve the request of zope.globalrequest when opening pages with mechanize. [deiferni]
- Also provide advice for available options in exception message. [lgraf]

1.19.1 (2015-08-20)

- Preserve radio-button input when filling forms with radio buttons. [deiferni]

1.19.0 (2015-07-31)

- Implement browser.click_on(tex) short cut for clicking links. [jone]
- Fix encoding error in assertion message when selecting a missing select option. [mbaechtold]

1.18.1 (2015-07-23)

- Fix GET form submission to actually submit it with GET. [jone]

1.18.0 (2015-07-22)

- Table: add new ".column" method for getting all cells of a column. [jone]

1.17.0 (2015-07-22)

- Add support for filling `collective.z3cform.datagridfield`. [jone, mbaechtold]

1.16.1 (2015-07-13)

- Autocomplete widget: extract URL from javascript. [jone]

1.16.0 (2015-07-08)

- Add image upload widget support (archetypes and dexterity). [jone]

1.15.0 (2015-05-07)

- Parse XML responses with XML parser instead of HTML parser. New methods for parsing the response: `parse_as_html`, `parse_as_xml` and `parse`. [jone]
- Add browser properties `contenttype`, `mimetype` and `encoding`. [jone]

1.14.6 (2015-04-17)

- Use `cssselect` in favor of `lxml.cssselect`. This allows us to use `lxml >= 3`. [jone]
- Added tests for z3c date fields. [phgross]

1.14.5 (2015-01-30)

- AutocompleteWidget: Drop query string from base URL when building query URL. [lgraf]

1.14.4 (2014-10-03)

- Widgets: test for sequence widget after testing for autocomplete widgets. Some widgets match both, autocomplete and sequence widgets. In this case we want to have the autocomplete widget. [jone]

1.14.3 (2014-10-02)

- Fix error with textarea tags without id-attributes. [jone]

1.14.2 (2014-09-29)

- Fix an issue with relative urls. [jone, deiferni]

1.14.1 (2014-09-26)

- Set the HTTP `REFERER` header correctly. [jone]

1.14.0 (2014-09-26)

- Add folder_contents page object. [jone]
- Update table methods with keyword arguments:
  - head_offset: used for stripping rows from the header
  - as_text: set to False for getting cell nodes

  [jone]

1.13.4 (2014-09-22)

- Filling selects: verbose error message when option not found. The available options are now included in the message. [jone]

1.13.3 (2014-09-02)

- Node.text: remove multiple spaces in a row caused by nesting. [jone]

1.13.2 (2014-08-06)

- Fix problems when filling forms which have checked checkedbox. [phgross]

1.13.1 (2014-07-15)

- Fix encoding problem on binary file uploads. [jone]

1.13.0 (2014-06-12)

- Add a Dexterity namedfile upload widget. [lgraf]

1.12.4 (2014-05-30)

- Fix python 2.6 support. [jone]

1.12.3 (2014-05-30)

- Fix z3cform choice collection widget to support Plone < 4.3. [jone]

1.12.2 (2014-05-29)

- Fix z3cform choice collection widget submit value. The widget creates hidden input fields on submit. [jone]

1.12.1 (2014-05-29)

- Fix error in z3cform choice collection widget when using paths. [jone]

1.12.0 (2014-05-29)

- Add a z3cform choice collection widget. This is used for z3cform List fields with Choice value_type. [jone]
- Add select field node wrapper with methods for getting available options. [jone]

1.11.4 (2014-05-22)

- browser.open(data): support multiple values for the same data name. The values can either be passed as a dict with lists as values or as a sequence of two-element tuples. [jone]

1.11.3 (2014-05-19)

- Fix browser.url regression when the previous request raised an exception. [jone]

1.11.2 (2014-05-17)

- Make NoElementFound exception message more verbose. When a *.first* on an empty result set raises a NoElementFound exception, the exception message now includes the original query. [jone]

1.11.1 (2014-05-17)

- Fix browser cloning regression in autocomplete widget "query". The cloned browser did no longer have the same headers / cookies, causing authenticated access to be no longer possible. [jone]
- New browser.clone method for creating browser clones. [jone]
- Update standard page objects to accept browser instace as keyword arguments. This makes it possible to use the page objects with non-standard browsers. [jone]

1.11.0 (2014-05-14)

- New browser.base_url property, respecting the <base> tag. [jone]
- New browser.debug method, opening the current page in your real browser. [jone]
- New browser.on method, a lazy variant of browser.open. [jone]
- New browser.reload method, reloading the current page. [jone]
- Improve requests library support:
  - Support choosing requests library, make Zope app setup optional. When no Zope app is set up, the `requests` library is set as default, otherwise `mechanize`.
  - Support form submitting with requests library.
  - Improve login and header support for equests library requests.
  - Add browser.cookies support for requests library requests.
  - Use requests library sessions, so that cookies and headers persist.
  - Automatically use "POST" when data is submitted.

  [jone]
- Login improvements:
  - Support passing member objects to browser.login(). The users / members are still expected to hav TEST_USER_PASSWORD as password.
  - Refactor login to use the new request header methods.

  [jone]
- Add request header methods for managing permanent request headers:
  - browser.append_request_header

  - browser.replace_request_header
  - browser.clear_request_header

  [jone]

- Refactor Form: eliminate class methods and do not use the global browser. This improves form support when running multiple browser instances concurrently.
  - Form.field_labels (class method) is now a instance property and public API.
  - Form.find_widget_in_form (class method) is removed and replaced with Form.find_widget (instance method).
  - Form.find_field_in_form (class method) is removed and replaced Form.get_field (instance method).
  - Form.find_form_element_by_label_or_name (class method) is removed and replaced with browser.find_form_by_field.
  - Form.find_form_by_labels_or_names (class method) is removed and replaced with browser.find_form_by_fields.
  - New Form.action_url property with the full qualified action URL.
  - Fix form action URL bug when using relative paths in combination with document-style base url.

  [jone]

- Fix wrapping input.label - this did only work for a part of field types. [jone]
- Fix UnicodeDecodeError in node string representation. [mathias.leimgruber]

1.10.0 (2014-03-19)

- Add NodeWrapper-properties:
  - innerHTML
  - normalized_innerHTML
  - outerHTML
  - normalized_outerHTML

  [jone, elioschmutz]

1.9.0 (2014-03-18)

- Add support for filling AT MultiSelectionWidget. [jone]

1.8.0 (2014-03-04)

- Add a `context` property to the browser with the current context (Plone object) of the currently viewed page. [jone]

1.7.3 (2014-02-28)

- Fix encoding problem in factories menu page object. The problem occured when having a "Restrictions..." entry in the menu. [jone]

1.7.2 (2014-02-25)

- Form: Support checking checkboxes without a value. Checkboxes without a value attribute are invalid but common. The default browser behavior is to fallback to the value "on". [jone]

1.7.0 (2014-02-03)

- ContentTreeWidget: support filling objects as values. [jone]

1.6.1 (2014-01-31)

- Implement *logout* on browser, logout before each login. [jone]

1.6.0 (2014-01-29)

- Add *cookies* property to the browser. [jone]

1.5.3 (2014-01-28)

- Fix multiple wrapping on browser.forms. [jone]

1.5.2 (2014-01-17)

- Implement archetypes datetime widget form filling. [jone]

1.5.1 (2014-01-07)

- Fix encoding problems when posting unicode data directly with Browser.open. [jone]
- Support form filling with bytestrings. [jone]
- Fix form filling with umlauts. [jone]
- Fix form fill for single select fields. [jone]

1.5.0 (2014-01-03)

- Implement AT file upload widget, because the label does not work. [jone]
- Implement file uploads. [jone]
- Add "headers" property on the browser. [jone]

1.4.0 (2013-12-27)

- Deprecate *normalized_text* method, replace it with *text* property. The *text* property is more intuitive and easier to remember. The *text* property has almost the same result as *normalized_text*, but it represents <*br/*> and <*p*> with single and double newlines respectively. *text* is to be the lxml *text* property, which contained the raw, non-recursive text of the current node and is now available as *raw_text* property. [jone]
- open_html: make debugging file contain passed HTML. [jone]
- Sequence widget: implement custom form filling with label support and validation. [jone]
- Sequence widget: add additional properties with inputs and options. [jone]

1.3.0 (2013-12-11)

- Implement "query" method on autocomplete widget. [jone]
- Implement form fill for z3cform datetime widget. [jone]
- Fix setting attributes on nodes when wrapped with NodeWrapper. [jone]
- Implement form fill for z3cform autocomplete widgets. [jone]
- Implement form fill for z3cform sequence widgets. [jone]
- Add `webdav` method for doing WebDAV requests with a ZServer. [jone]

1.2.0 (2013-11-24)

- Add *open_html* method to browser object, allowing to pass in HTML directly. [jone]

1.1.0 (2013-11-07)

- Add dexterity page object, refactor z3cform page object. [jone]
- Add table nodes with helpers for table testing. [jone]
- Merging "Nodes" lists returns a new "Nodes" list, not a "list". [jone]
- Show containing elements in string representation of "Nodes" list. [jone]
- Fix direct child selection with CSS (node.css(">tag")). [jone]
- Add a `recursive` option to `normalized_text`. [jone]

1.0.2 (2013-10-31)

- When normalizing whitespaces, do also replace non-breaking spaces. [jone]

1.0.1 (2013-10-31)

- Add `first_or_none` property to `Nodes`. [jone]

1.0.0 (2013-10-28)

- Initial implementation. [jone]

CHAPTER 6

Links

- Source code on github: https://github.com/4teamwork/ftw.testbrowser
- Releases on pypi: https://pypi.python.org/pypi/ftw.testbrowser
- Issues on github: https://github.com/4teamwork/ftw.testbrowser/issues
- Continuous integration: https://jenkins.4teamwork.ch/search?q=ftw.testbrowser

CHAPTER 7

Indices and tables

- genindex
- modindex
- search