Selenium and other **UI-testing tools** in Python

Igor Davydenko
2012 Kyiv.py#8

TDD is a **huge myth** :(

- You **shouldn't** always write test before code
- You **couldn't** always write test before code
- The best practice for me is **lazy TDD**, when I commit my tests before fixes, but can write code and tests at the same time
- In most projects we also need to have *integrational tests, tests which works with real data, UI tests, specific tests for some platforms, etc*

BTW, all we need is tests

- **unittest2** is a part of Python stdlib
- **nosetests** doesn't die! And there is **py.test** available if you want something more
- Write tests is not a main part, main part is running tests!
- Don't forget about continuous integration systems, like **Jenkins**, **Travis**, etc

Running tests

unittest

```
$ python project/tests.py
$ find project/ -name tests.py -exec python {} \;
```

unittest2

```
$ python -m unittest project.tests
$ python -m unittest discover -s project/
```

nosetests

```
$ nosetests -v project/tests.py
$ nosetests -v -w project/
```

nosetests real usage

```
$ nosetests -x -v -w project/
$ nosetests --with-plugin -w project/
```

Start the fun

- Test project: https://github.com/playpauseandstop/learnpython.in.ua
- Framework: **Flask**
- Database: **FlatPages**
- Test Runner: **nosetests**

First test case

```
from unittest import TestCase

from flask import url_for

from learnpython.app import app


class TestViews(TestCase):

    def setUp(self):
        app.testing = True
        self.client = app.test_client()

        self._ctx = app.test_request_context()
        self._ctx.push()

    def tearDown(self):
        self._ctx.pop()

    def test_index(self):
        index_url = url_for('page', name='index')
        index_url_ext = url_for('page', name='index', _external=True)

        response = self.client.get('/')
        self.assertEqual(response.status_code, 301)
        self.assertEqual(response.headers['Location'], index_url_ext)

        response = self.client.get(index_url)
        self.assertEqual(response.status_code, 200)
        self.assertIn('Learn Python', response.data)

    def test_static(self):
        url = url_for('static', filename='css/screen.css')
        response = self.client.get(url)
        self.assertEqual(response.status_code, 200)

        url = url_for('static', filename='does_not_exists.exe')
        response = self.client.get(url)
        self.assertEqual(response.status_code, 404)
```

Improve tests with **Flask-Testing**

- GitHub: https://github.com/jarus/flask-testing/
- Don't need to do routine, only to define `create_app` method
- Bunch of `assert*` functions (`assert200`, `assert404`, `assertRedirects`, `assertStatus`, `assertTemplateUsed`)
- Helpers to test JSON responses (`response.json` property)
- Support of Twill browser (`Twill` context manager)

Updated test case

```
from flask import url_for
from flask.ext.testing import TestCase

from learnpython.app import app


class TestViews(TestCase):

    def create_app(self):
        app.testing = True
        return app

    def test_index(self):
        index_url = url_for('page', name='index')

        response = self.client.get('/')
        self.assertRedirects(response, index_url)
        self.assertStatus(response, 301)

        response = self.client.get(index_url)
        self.assert200(response)
        self.assertIn('Learn Python', response.data)

    def test_static(self):
        url = url_for('static', filename='css/screen.css')
        response = self.client.get(url)
        self.assert200(response)

        url = url_for('static', filename='does_not_exists.exe')
        response = self.client.get(url)
        self.assert404(response)
```

Twill is dead, but we still using it

- Old old site: http://twill.idyll.org/
- Latest version, 0.9, released *Thursday, December 27th, 2007*
- But Twill commands still rule for easyfying browser-like tasks
- Submitting forms is easy as "`fv(1, 'name', 'value'); submit(); code(200); url(url_regex);`"
- Supported by Flask-Testing, has tons of connectors for Django, other Python frameworks

Twill test case

```
from flask import url_for
from flask.ext.testing import TestCase, Twill
from twill.commands import code, go, find, url

from learnpython.app import app


class TestViews(TestCase):

    def create_app(self):
        app.testing = True
        return app

    def test_index(self):
        with Twill(self.app, port=8080) as t:
            index_url = t.url(url_for('page', name='index'))

            go(t.url('/'))
            code(200)
            url(index_url)
            find('Learn Python')

    def test_static(self):
        with Twill(self.app, port=8080) as t:
            url = t.url(url_for('static', filename='css/screen.css'))
            go(url)
            code(200)

            url = url_for('static', filename='does_not_exists.exe')
            go(url)
            code(404)
```

But maybe just try **mechanize**

- Site: http://wwwsearch.sourceforge.net/mechanize/
- Core for Twill's browser, like `Werkzeug` for `Flask`
- No easy way to intersept WSGI applications
- No methods for matching text in responses
- Looks more like older `requests`, than newer `twill`

Flask-Testing issues

- Pass config values to app is not an easy task, they could be just test case attributes,

```
class TestViews(TestCase):
    CSRF_ENABLED = False
    TESTING = True
```

- With `Twill(self.app, port=8080) as t` context manager only `Browser` instance available, not all `twill.commands`

- Matching something in response without Twill is painful task

Is that all choices?
Nah, welcome **WebTest**

- Site: http://webtest.pythonpaste.org/
- Support testing any WSGI app with wrapping it into `TestApp` instance
- Support different response body's parsing (`BeautifulSoup`, `ElementTree`, `lxml`, `PyQuery`, `json`)
- Work with forms is same easy as in Twill
- Response compatible with Flask's default response
- Highly powerful tool for web-testing without real browser

WebTest test case

```
from flask import url_for
from flask.ext.testing import TestCase
from webtest import TestApp

from learnpython.app import app


class TestViews(TestCase):

    def assertRedirects(self, response, location):
        location = ':80{0}'.format(location)
        super(TestViews, self).assertRedirects(response, location)

    def create_app(self):
        app.testing = True
        self.webtest = TestApp(app)
        return app

    def test_index(self):
        index_url = url_for('page', name='index')

        response = self.webtest.get('/', status=301)
        self.assertRedirects(response, index_url)

        response = self.webtest.get(index_url)
        self.assert200(response)
        self.assertEqual(
            response.pyquery('a[href="{0}"]'.format(index_url)).text(),
            'Learn Python'
        )

    def test_static(self):
        url = url_for('static', filename='css/screen.css')
        response = self.webtest.get(url, status=200)

        url = url_for('static', filename='does_not_exist.exe')
        response = self.webtest.get(url, status=404)
```

Problems with web UI unit testing

- Only Twill or WebTest are good for getting browser-like approach, but Twill is smells outdated and WebTest notations sometimes looks monstrous and weird

- Built-in Flask and Django test clients isn't any good for testings sites' UI and more designed for testing API responses

- Database-specific issues, you need to use fixtures, mocks, other magic tricks before executing the test

Welcome, **Selenium**!

- *Selenium automates browsers*, http://seleniumhq.org/
- Portable software testing framework for web applications
- Provides a test domain-specific language Selenese, can write tests in C#, Java, Perl, PHP, Python and Ruby
- Selenium WebDriver runs tests in Firefox, Chrome, Internet Explorer, Opera browsers
- Selenium deploys on Windows, Linux and Mac OS

What Selenium can?

- Full interaction with browser (back, forward, reload, work with cookies, saving screenshots, maximize/close windows)

- Finding elements by class name, css selector, id, link text, name, tag name, xpath

- Execute Selenium command, JavaScript or async JavaScript in browser

- Setting timeouts for page load, script load to test how fast page loaded, setting window dimensions before testing

How it works? Part 1

- Install Selenium Python bindings, `$ pip install selenium`
- Create regular unit test case (in most cases we don't need to use specific test case class)
- Init `webdriver.Firefox` or other supported WebDriver instance (particullary in `setUp` method)
- Use `browser.open`, `browser.click`, `browser.find_element` and other methods in tests
- Quit all browser windows with `browser.quit` in `tearDown`

How it works? Part 2

- Save tests to `project/tests/test_selenium.py`
- `(env)$ wget http://selenium.googlecode.com/files/selenium-server-standalone-2.25.0.jar`
- `(env)$ java -jar selenium-server-standalone-2.25.0.jar &`
- `(env)$ python app.py $SELENIUM_HOST:$SELENIUM_PORT &`
- ```
  (env)$ SELENIUM_URL=http://$SELENIUM_HOST:$SELENIUM_PORT \
         nosetests -v project/test_selenium.py
  ```
- `(env)$ kill-python-server`
  ```

Selenium test case

```
import time

from unittest import TestCase

from flask import url_for
from selenium import webdriver

from learnpython.app import app


class TestViewsWithSelenium(TestCase):

    def setUp(self):
        self.browser = webdriver.Firefox()
        app.testing = True
        self._ctx = app.test_request_context()
        self._ctx.push()

    def tearDown(self):
        self._ctx.pop()
        self.browser.quit()

    ...

    def test_index(self):
        index_url = self.url('page', name='index')

        self.browser.get(self.url('/'))
        self.assertEqual(self.browser.current_url, index_url)
        time.sleep(1)

        element = self.browser.find_element_by_link_text('Learn Python')
        self.assertEqual(element.get_attribute('href'), index_url)

    def test_static(self):
        self.browser.get(self.url('static', filename='css/screen.css'))
        self.browser.get(self.url('static', filename='does_not_exists.exe'))
```

Selenium Grid

- http://selenium-grid.seleniumhq.org/
- In-browser tests are slow and not cover all problems
- Hardware is cheap
- Selenium Grid allows you to run multiple instances of Selenium WebDrivers in parallel
- Supports Amazon EC2 out-of-box

Selenium + Jenkins = success story

- http://www.slideshare.net/MaxKlymyshyn/testing-with-jenkins-selenium-and-continuous-deployment
- Install and configure Selenium Grid at EC2 test instance
- Deploy each commit, tested by Jenkins, as subdomain in main EC2 instace
- Run selenium tests for each deployed commit and for each necessary browser
- Remove deployed commit from EC2 instance

But, say hello to **Windmill** too

- http://www.getwindmill.com/
- They said Windmill is a web testing tool designed to let you painlessly automate and debug your web application
- By their idea, Windmill could do anything what Selenium does, but without standalone server
- In real world, I don't understand how exactly it works :)

Windmill test case

```
from unittest import TestCase

from windmill.authoring import WindmillTestClient, setup_module, \
    teardown_module

from learnpython.app import app


class TestViewsWithWindmill(TestCase):

    def setUp(self):
        app.testing = True
        self.client = WindmillTestClient(__name__)
        self._ctx = app.test_request_context()
        self._ctx.push()

    def tearDown(self):
        self._ctx.pop()

    def test_index(self):
        self.client.open()
```

Is this all? Nah, not so fast

- There are a lot more Python UI testing tools
- **Splinter** http://splinter.cobrateam.info/
- Successor of Twill, full working Python answer to Selenium
- Can do practically anything which Selenium can and doesn't need standalone server

Splinter test case

```
from unittest import TestCase

from splinter import Browser
from splinter.request_handler.status_code import HttpResponseError

from learnpython.app import app


class TestViewsWithSplinter(TestCase):

    def setUp(self):
        app.testing = True
        self.browser = Browser()
        self._ctx = app.test_request_context()
        self._ctx.push()

    def tearDown(self):
        self._ctx.pop()
        self.browser.quit()

    ...

    def test_index(self):
        index_url = self.url('page', name='index')

        self.browser.visit(self.url('/'))
        self.assertEqual(self.browser.url, index_url)
        self.assertEqual(self.browser.title, 'Learn Python in Kyiv, Ukraine')

        link = self.browser.find_link_by_text('Learn Python')
        self.assertEqual(len(link), 1)
        self.assertEqual(link[0].text, 'Learn Python')

    def test_static(self):
        self.browser.visit(self.url('static', filename='css/screen.css'))
        self.assertEqual(self.browser.status_code.code, 200)

        self.assertRaises(
            HttpResponseError,
            self.browser.visit,
            self.url('static', filename='does_not_exist.exe')
        )
```

And something more

- **Spynner** https://github.com/makinacorpus/spynner
- **SST** http://testutils.org/sst/
- **Robot Framework** http://code.google.com/p/robotframework/

So, what's the deal?

- **Splinter** is more Pythonic way to test UI
- **Selenium** is more used way to test UI
- **Windmill** is bunch of something, which does something, I don't know what

Don’t you forget on **BDD**?

- BDD is cool, but needs some magic too
- **Lettuce** http://lettuce.it/
  Write features, fill it with scenarios, each scenario contains commands, code commands in Python
- **Behave** http://pypi.python.org/pypi/behave
  Same idea

Ideal UI tests
shouldn't write in Python

- UI tests are not nice at all
- They don't fit to PEP8 and Python style
- UI tests better write in JS, Selenese or something not called Python
- I don't want to write UI tests using Selenium, Splinter, etc
  I want to write UI tests using Flask-Testing, WebTest, etc

Questions?

Igor Davydenko
http://igordavydenko.com/
https://github.com/playpauseandstop