

Acceptance Testing for Plone

Using Funittest

Plone Conference 2007

Maik Röder

Yaco Sistemas

maikroeder@gmail.com

Acceptance Testing

- Agile software development methodology
- Extreme Programming
- Functional testing of a user story during the implementation phase
- Black box system tests
- Regression tests prior to a production release.

Requirements

Or, what I expect from a functional testing system

- Use case centered
- High level
- Readability
- Reusability
- Extensibility
- Modularization
- Simple elegance

Acceptance Testing for Plone with Funittest

- Write acceptance test first
 - Test driven development
- Guarantee the quality of your Plone sites
 - Untested sites are broken sites
- Run in-browser acceptance tests
 - Selenium Remote Control

- Real TestBrowser
- Extensive library of reusable scripts, verbs, scenarios and tests

History

- Selenium
- PloneSelenium
- PLIP #100: Integrate Selenium for functional testing
- Funittest
- Funittest sprint

Development with Funittest

- Documentation driven development
- Start with the Use Cases
 - Tell a story of what the user does step by step
- Write new use case scenarios or extensions
 - Developers may find documentation and think about other extensions
- Write high-level, domain-specific, vocabulary for each new user action
- Write new low level methods for vocabulary
- Write new functional tests reusing the scenarios
- Write new verification code outside of tests

Presets

- Prepare a site for tests
- Add sample users to the site for testing
- Make changes to the site settings to facilitate testing
- Presets for Plone 3 and Plone 2.5 exist
- Simple Python file with a preset method
- Presets are run against a Plone site to make it establish data providers

```
def preset():  
  
    load_funittest_model("funittest.lib.plone301.cmfplone")  
  
    interpreter.setBrowserURL("http://localhost:8080/Plone")  
    dataprovider.cmfplone.user.current = ['admin',  
    'samplemanager', 'samplemember']  
    return [dataprovider.cmfplone.user]
```

Custom presets

- Each of your Plone projects should have a custom preset
- Have a look at the example Custom preset for implementing your own
- Run a preset against your Plone site:

```
python preset.py --preset myproject
```

- Plone site is ready for testing
- Does not (yet) use Plone Policy Product

Data Providers

- Test fixtures
- Collection of example data for the tests
- Can contain a method for establishing the data used in the preset
- Simple dictionary of dictionaries

```
>>> from funittest import dataprovider
>>> dataprovider.cmfplone.user.get("samplemember")
{'visible_ids': True, 'roles': ['Member'], 'id':
'samplemember', 'groups': [], 'fullname': 'Sample
Member', 'password': 'seleniumtest', 'email':
'test@example.com', 'password_confirm': 'seleniumtest'}
```

Tests

- Tests are reusable
- Tests are generic
- Reuse tests on all of your new Plone sites
- Add a generic test, use it everywhere
- Execute all tests:

```
python2.4 test.py
...
-----
Ran 3 tests in 0.148s

OK
```

- Uses standard Python unittest library

Tests depend on the whole functional test stack

- Tests depend on all the other elements of the test stack
- Use a custom preset to decide what parts your stack should use

- Failing tests tell you
 - what to change in the stack
 - what to fix in your Plone site
- Run tests again until all tests pass
- Run tests again tomorrow to catch regressions

Execute a single test

- Addable content test:
 - Test that all expected content types are addable

```
python2.4 test.py --preset custom -t addablecontent
['PloneGlossary not in list of addable types', ...]
...
```

- Structure of a test class

```
class AddableContent(Test):
    def setUp(self):
        # No Setup here
    def step_1(self):
        # Verify expected addable types at Plone site
root
    def test(self):
        self.expect_ok(1)
```

- Tests are first class citizens

Tests are testable

- Make tests fail to prove that they do something useful

```
python2.4 test.py --preset custom -t addablecontent -e
1b
```

```
class AddableContent(Test):
    ...
    def test_1b(self):
        # Add bogus content type
        self.expect_ko(1)
        # Remove bogus content type
        self.expect_ok(1)
```

- Forces the developer to write the verification code
- Verification can vary from site to site
- Verification can be effective on one site, and do nothing on another
- When the verification code changes for whatever reason, the test of the test fails

Verification

- Verification steps
 - Catch state before
 - Change expected state
 - Execute verb
 - Compare expected state and real state
- Verification is factored out

```
def submit(self, user):
    element = "//dl[@class='portalMessage error']"
    if interpreter.is_element_present(element):
        interpreter.verifyNotVisible(element)
```

- Comprehensive verification
 - Systematically verify general state on each action, not just in an isolated test

Use Cases

- Put yourself into the position of the user
- What are the actions the user should be able to do?
- What is the main scenario?
- What are the alternative scenarios?
- What errors are possible?
- Write extension points

'Register a new user' Use Case

- Registration of a new user to the Plone site
 1. Access the registration form
 2. Fill in the registration form
 3. Submit the registration form
 4. Directly log in to the site

Find verbs in use case

- Taking the point of view of the user
- What is the user action in each step?
- "access registration form", "fill in form", "submit", "login directly"
- Group verbs in different logical functional models, like Application, Content, Folder, Navigation, Search
- New domain "register"

Main Scenario Steps

- Write down scenario steps using the verbs from the new "register" domain
- Define the steps of the main scenario calling the logical functional model

```
def step_1(self):
    logical.cmfpone.register.access(self._user)

def step_2(self):
    logical.cmfpone.register.fill(self._user)

def step_3(self):
    logical.cmfpone.register.submit(self._user)

def step_4(self):
    logical.cmfpone.register.direct_login(self._user)
```

Scenario Schemas

- Schema contains scenario parameters

```
schema =
Schema({"user":dataprovder.cmfpone.user})
```

- Make parameters available to scenario code

Main Scenario

- Your main scenario is expected to work ok:

```
def scenario(self):
    """
    User registers
    """
    self.expect_ok(1,2,3,4)
```

- Execute scenario

```
python2.4 scenario.py --preset custom -s register
```

Extension Points

- You define the extension points
- Fail at first
- If recovery is possibly, make scenario finish ok

```
def scenario_3a(self):
```

```

    """
    User enters password different from the
    confirmation password
    """
    password = self._user['password']
    self._user['password']='differentfirstpassword'
    self.expect_ko(1,2,3)
    # Recover from the error by filling in the correct
    password this time
    self._user['password']=password
    self.expect_ok(2,3,4)

```

Using Scenarios from code

- Scenarios can be used from Python code like this

```
scenarios.cmfplone.register(user=user)
```

- Scenarios can be tried

```
scenarios.cmfplone.installproduct.try_scenario()
```

- Scenario steps can be tried

```
scenarios.cmfplone.addcontent.try_step(1)
```

Logical Functional Model

- Define the logical functional model verbs

```

def access(self, user):
    physical.cmfplone.register.access(user)

def fill(self, user):
    physical.cmfplone.register.fill(user)

def submit(self, user):
    physical.cmfplone.register.submit(user)

def direct_login(self, user):
    physical.cmfplone.register.direct_login(user)

```

Physical Model

- Implement the physical model methods

```

def access(self, user):
    interpreter.open('/join_form')

```

```

def fill(self, user):
    forms = physical.cmfplone.forms.getForms()
    form =
forms.getByLocator("//form[@action='join_form']")
    values = []
    values.append( {'id':'fullname',
                    'value':user['fullname']} )
    values.append( {'id':'username',
                    'value':user['id']} )
    values.append( {'id':'email',
                    'value':user['email']} )
    values.append( {'id':'password',
                    'value':user['password']} )
    values.append( {'id':'password_confirm',
                    'value':user['password_confirm']} )
    form.fillForm(values)

def submit(self, user):
    interpreter.clickAndWait("form.button.Register")

def direct_login(self, user):
    interpreter.annotate("Login after registration")
    interpreter.clickAndWait("//input[@value='Log
in']")

```

Get Funittest

- Collective
- `svn co https://svn.plone.org/svn/collective/funittest/trunk funittest`
- `cd funittest`
- `less README.txt`
- Interactive browser demo of using Crunchy
- `cd funittest/doc`
- `python crunchy.py`

Literature and Resources

- Funittest:
 - Plone Conference 2007 Sprint Page
 - <http://www.openplans.org/projects/plone-conference-2007/funittest>
- The Braidy Tester:
 - Functional Test Stack Articles
 - <http://www.thebraidytester.com/>
- Article by Jennitta Andrea
 - Brushing Up On Functional Test Effectiveness
 - http://www.stickyminds.com/s.asp?F=S9937_ART_2
- Book by Alistair Cockburn
 - Writing Effective Use Cases
 - http://alistair.cockburn.us/index.php/Resources_for_writing_use_cases

)