plone.act Documentation

Release 1.0a1

Plone Foundation

April 06, 2013

CONTENTS

Warning: **plone.act** is deprecated.
Please, use http://pypi.python.org/pypi/plone.app.robotframework instead.

plone.act and its documentation gives you everything to get started in writing and executing functional Selenium tests (including acceptance tests) for your Plone add-on.

plone.act performs functional testing by using two testing frameworks: Robot Framework and Selenium.

Robot Framework is a generic test automation framework for acceptance testing and acceptance test-driven development (ATDD), even for behavior driven development (BDD). It has easy-to-use plain text test syntax and utilizes the keyword-driven testing approach. Selenium is a web browser automation framework that exercises the browser as if the user was interacting with the browser.

CHAPTER

ONE

START HERE

1.1 Write a robot test for a new Plone add-on

This is a minimal tutorial for getting started with writing functional Selenium tests for a new Plone add-on with Robot Framework.

1.1.1 Install Templer

At first, we should have an add-on to test with. For creating a new add-on, we use Templer.

1. Create a directory for a Templer-buildout and move there:

```
$ mkdir templer-buildout
$ cd templer-buildout
```

2. Create a file `templer-buildout/buildout.cfg` for Templer-installation with:

```
[buildout]
parts = templer

[templer]
recipe = zc.recipe.egg
eggs =
    templer.core
    templer.plone
```

3. Download a bootstrap for running the buildout:

```
$ curl -O http://python-distribute.org/bootstrap.py
```

4. Bootstrap and run the buildout:

```
$ python bootstrap.py --distribute
$ bin/buildout
Installing templer.
Generated script '/.../templer-buildout/bin/templer'.
```

5. Return back to the parent directory:

```
$ cd ..
```

1.1.2 Create a new product

Once we have Templer installed, we create a Plone add-on product by entering `templer-buildout/bin/templer plone_basic` and answering to the upcoming questions.

We must make sure to answer `True` for the question:

```
Robot Tests (Should the default robot test be included) [False]: True
```

Once we have answered for all the questions, our add-on template is ready:

```
$ templer-buildout/bin/templer plone_basic

plone_basic: A package for Plone add-ons

This template creates a package for a basic Plone add-on project with
a single namespace (like Products.PloneFormGen).

To create a Plone project with a name like 'collective.geo.bundle'
(2 dots, a 'nested namespace'), use the 'plone_nested' template.

If you are trying to create a Plone *site* then the best place to
start is with one of the Plone installers.  If you want to build
your own Plone buildout, use one of the plone'N'_buildout templates


This template expects a project name with 1 dot in it (a 'basic
namespace', like 'foo.bar').

Enter project name (or q to quit): my.product

If at any point, you need additional help for a question, you can enter
'?' and press RETURN.

Expert Mode? (What question mode would you like? (easy/expert/all)?) ['easy']:
Version (Version number for project) ['1.0']:
Description (One-line description of the project) ['']:
Register Profile (Should this package register a GS Profile) [False]:
Robot Tests (Should the default robot test be included) [False]: True
Creating directory ./my.product
Replace 1019 bytes with 1378 bytes (2/43 lines changed; 8 lines added)
Replace 42 bytes with 119 bytes (1/1 lines changed; 4 lines added)
```

1.1.3 Bootstrap and run buildout

Before we continue, now is a good time to run bootstrap and buildout to get the development environment ready:

```
$ python bootstrap.py --distribute
$ bin/buildout
```

1.1.4 Run the default tests

Templer does create a couple of example tests for us – one of them being a robot test.

We can list the available tests with:

```
$ bin/test --list-tests
Listing my.product.testing.MyproductLayer:Functional tests:
  Plone site (robot_test.txt) #start
Listing my.product.testing.MyproductLayer:Integration tests:
  test_success (my.product.tests.test_example.TestExample)
```

And run the example robot test with:

```
$ bin/test -t robot_
Running my.product.testing.MyproductLayer:Functional tests:
  Set up plone.testing.zca.LayerCleanup in 0.000 seconds.
  Set up plone.testing.z2.Startup in 0.237 seconds.
  Set up plone.app.testing.layers.PloneFixture in 8.093 seconds.
  Set up my.product.testing.MyproductLayer in 0.178 seconds.
  Set up plone.testing.z2.ZServer in 0.503 seconds.
  Set up my.product.testing.MyproductLayer:Functional in 0.000 seconds.
  Running:

  Ran 1 tests with 0 failures and 0 errors in 2.588 seconds.
Tearing down left over layers:
  Tear down my.product.testing.MyproductLayer:Functional in 0.000 seconds.
  Tear down plone.testing.z2.ZServer in 5.251 seconds.
  Tear down my.product.testing.MyproductLayer in 0.004 seconds.
  Tear down plone.app.testing.layers.PloneFixture in 0.087 seconds.
  Tear down plone.testing.z2.Startup in 0.006 seconds.
  Tear down plone.testing.zca.LayerCleanup in 0.005 seconds.
```

1.1.5 About functional test fixture

Functional Selenium tests require a fully functional Plone-environment.

Luckily, with plone.app.testing we can easily define a custom test fixture with Plone and our own add-on installed.

With Templer, both the base fixture and the functional test fixtures have already been defined in `my.product/src/my/product/testing.py`. The latter with:

```
from plone.app.testing import FunctionalTesting

...

MY_PRODUCT_FUNCTIONAL_TESTING = FunctionalTesting(
    bases=(MY_PRODUCT_FIXTURE, z2.ZSERVER_FIXTURE),
    name="MyproductLayer:Functional"
)
```

1.1.6 Create a new robot test suite

Robot tests are written as text files, which are called test suites.

It's good practice, with Plone, to prefix all robot test suite files with `robot_`. This makes it easier to both exclude the robot tests (which are usually very time consuming) from test runs or run only the robot tests.

Write an another robot tests suite `my.product/src/my/product/tests/robot_hello.txt`:

```
*** Settings ***

Library  Selenium2Library  timeout=10  implicit_wait=0.5
```

```
Suite Setup  Start browser
Suite Teardown  Close All Browsers

*** Variables ***

${BROWSER} =  Firefox

*** Test Cases ***

Hello World
    [Tags]  hello
    Go to  http://localhost:55001/plone/hello-world
    Page should contain  Hello World!

*** Keywords ***

Start browser
    Open browser  http://localhost:55001/plone/  browser=${BROWSER}
```

Note: Defining browser for `Open browser` keyword as a variable makes it easy to run the test later with different browser.

1.1.7 Register the suite for zope.testrunner

To be able to run Robot Framework test suite with zope.testrunner and on top of our add-ons functional test fixture, we need to

1. wrap the test suite into properly named Python unittest test suite
2. assign our functional test layer for all the test cases.

We do this all by simply adding our new robot test suite into `my.product/src/my/product/tests/test_robot.py`:

```
from my.product.testing import MY_PRODUCT_FUNCTIONAL_TESTING
from plone.testing import layered
import robotsuite
import unittest


def test_suite():
    suite = unittest.TestSuite()
    suite.addTests([
        layered(robotsuite.RobotTestSuite("robot_test.txt"),
                layer=MY_PRODUCT_FUNCTIONAL_TESTING),
        layered(robotsuite.RobotTestSuite("robot_hello_world.txt"),
                layer=MY_PRODUCT_FUNCTIONAL_TESTING)
    ])
    return suite
```

Note that `test_`-prefix in the filename of `test_robot.py` is required for **zope.testunner** to find the test suite.

1.1.8 List and filter tests

Run `bin/test` (**zope.testrunner**) with `--list-tests`-argument to see that our test is registered correctly:

```
$ bin/test --list-tests
Listing my.product.testing.MyproductLayer:Functional tests:
  Plone site (robot_test.txt) #start
  Hello_World (robot_hello_world.txt) #hello
Listing my.product.testing.MyproductLayer:Integration tests:
  test_success (my.product.tests.test_example.TestExample)
```

Experiment with `-t`-argument to filter testrunner to find only our robot test:

```
$ bin/test -t robot_ --list-tests
Listing my.product.testing.MyproductLayer:Functional tests:
  Plone site (robot_test.txt) #start
  Hello_World (robot_hello_world.txt) #hello
```

or everything else:

```
$ bin/test -t \!robot_ --list-tests
Listing my.product.testing.MyproductLayer:Integration tests:
  test_success (my.product.tests.test_example.TestExample)
```

We can also filter robot tests with tags:

```
$ bin/test -t \#hello --list-tests
Listing my.product.testing.MyproductLayer:Functional tests:
  Hello_World (robot_hello_world.txt) #hello
```

1.1.9 Run (failing) test

After the test has been written and registered, it can be run normally with `bin/test`.

The run will fail, because the test describes an unimplemented feature:

```
$ bin/test -t \#hello

Running my.product.testing.MyproductLayer:Functional tests:
  Set up plone.testing.zca.LayerCleanup in 0.000 seconds.
  Set up plone.testing.z2.Startup in 0.217 seconds.
  Set up plone.app.testing.layers.PloneFixture in 7.643 seconds.
  Set up my.product.testing.MyproductLayer in 0.026 seconds.
  Set up plone.testing.z2.ZServer in 0.503 seconds.
  Set up my.product.testing.MyproductLayer:Functional in 0.000 seconds.
  Running:
    1/1 (100.0%)
==============================================================================
Robot Hello World
==============================================================================
Hello World                                                           | FAIL |
Page should have contained text 'Hello World!' but did not
------------------------------------------------------------------------------
Robot Hello World                                                     | FAIL |
1 critical test, 0 passed, 1 failed
1 test total, 0 passed, 1 failed
==============================================================================
Output:  /.../my.product/parts/test/robot_hello_world/Hello_World/output.xml
Log:     /.../my.product/parts/test/robot_hello_world/Hello_World/log.html
Report:  /.../my.product/parts/test/robot_hello_world/Hello_World/report.html
```

```
Failure in test Hello World (robot_hello_world.txt) #hello
Traceback (most recent call last):
  File "/.../unittest2-0.5.1-py2.7.egg/unittest2/case.py", line 340, in run
    testMethod()
  File "/.../eggs/robotsuite-1.0.2-py2.7.egg/robotsuite/__init__.py", line 317, in runTest
    assert last_status == 'PASS', last_message
AssertionError: Page should have contained text 'Hello World!' but did not


  Ran 1 tests with 1 failures and 0 errors in 3.632 seconds.
Tearing down left over layers:
  Tear down my.product.testing.MyproductLayer:Functional in 0.000 seconds.
  Tear down plone.testing.z2.ZServer in 5.282 seconds.
  Tear down my.product.testing.MyproductLayer in 0.003 seconds.
  Tear down plone.app.testing.layers.PloneFixture in 0.084 seconds.
  Tear down plone.testing.z2.Startup in 0.006 seconds.
  Tear down plone.testing.zca.LayerCleanup in 0.004 seconds.
```

1.1.10 Create an example view

Create view described in the test by registering a template into `my.product/src/my/product/configure.zcml`:

```
<configure
    xmlns="http://namespaces.zope.org/zope"
    xmlns:five="http://namespaces.zope.org/five"
    xmlns:browser="http://namespaces.zope.org/browser"
    xmlns:i18n="http://namespaces.zope.org/i18n"
    xmlns:genericsetup="http://namespaces.zope.org/genericsetup"
    i18n_domain="my.product">

  <five:registerPackage package="." initialize=".initialize" />

  <browser:page
      name="hello-world"
      for="Products.CMFCore.interfaces.ISiteRoot"
      template="hello_world.pt"
      permission="zope2.View"
      />

  <!-- -*- extra stuff goes here -*- -->

</configure>
```

And writing the template into `my.product/src/my/product/hello_world.pt`:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
      xmlns:tal="http://xml.zope.org/namespaces/tal"
      xmlns:metal="http://xml.zope.org/namespaces/metal"
      xmlns:i18n="http://xml.zope.org/namespaces/i18n"
      lang="en"
      metal:use-macro="context/main_template/macros/master"
      i18n:domain="plone">
<body>

<metal:content-core fill-slot="content-core">
    <metal:content-core define-macro="content-core">
      <p>Hello World!</p>
```

```
    </metal:content-core>
</metal:content-core>

</body>
</html>
```

1.1.11 Run (passing) test

Re-run the test to see it passing:

```
$ bin/test -t \#hello
Running my.product.testing.MyproductLayer:Functional tests:
  Set up plone.testing.zca.LayerCleanup in 0.000 seconds.
  Set up plone.testing.z2.Startup in 0.220 seconds.
  Set up plone.app.testing.layers.PloneFixture in 7.810 seconds.
  Set up my.product.testing.MyproductLayer in 0.027 seconds.
  Set up plone.testing.z2.ZServer in 0.503 seconds.
  Set up my.product.testing.MyproductLayer:Functional in 0.000 seconds.
  Running:

  Ran 1 tests with 0 failures and 0 errors in 2.604 seconds.
Tearing down left over layers:
  Tear down my.product.testing.MyproductLayer:Functional in 0.000 seconds.
  Tear down plone.testing.z2.ZServer in 5.253 seconds.
  Tear down my.product.testing.MyproductLayer in 0.004 seconds.
  Tear down plone.app.testing.layers.PloneFixture in 0.085 seconds.
  Tear down plone.testing.z2.Startup in 0.006 seconds.
  Tear down plone.testing.zca.LayerCleanup in 0.004 seconds.
```

1.1.12 Test reports

Robot Framework generates high quality test reports with screenshots of failing tests as:

`my.product/parts/tests/robot_report.html` Overview of the test results.

`my.product/parts/tests/robot_log.html`: Detailed log for every test with screenshots of failing tests.

1.2 Write a robot test for an existing Plone add-on

This is a tutorial for getting started with writing functional Selenium tests for an existing Plone add-on with Robot Framework.

Let's assumpt that we have an add-on **my.product**.

1.2.1 Update requirements

At first, we need to fix our product to require all the necessary dependencies for running Robot Framework tests.

To fix our dependencies, we update `my.product/setup.py` with:

```
extras_require={'test': ['plone.app.testing[robot]]},
```

Note: When testing with Plone version less than 4.3, we must pin the version of **plone.app.testing** into `buildout.cfg`.

Update `my.product/buildout.cfg` with:

```
[buildout]
extends =
    ...
    versions.cfg
```

And create `my.product/versions.cfg` with:

```
[versions]
plone.app.versions = 4.2.2
```

1.2.2 Bootstrap and run buildout

Before we continue, now is a good time to run bootstrap and buildout to get the development environment ready:

```
$ python bootstrap.py --distribute
$ bin/buildout
```

1.2.3 Define functional test fixture

Functional Selenium tests require a fully functional Plone-environment.

Luckily, with plone.app.testing we can easily define a custom test fixture with Plone and our own add-on installed.

After the base fixture has been created (by following plone.app.testing documentation) we only need to define a functional testing fixture, which adds a fully functional ZServer to serve a Plone sandbox with our add-on.

Update `my.product/src/my/product/testing.py` with:

```
from plone.app.testing import FunctionalTesting


MY_PRODUCT_FUNCTIONAL_TESTING = FunctionalTesting(
    bases=(MY_PRODUCT_FIXTURE, z2.ZSERVER_FIXTURE),
    name="MyproductLayer:Functional"
)
```

1.2.4 Create a robot test suite

Robot tests are written as text files, which are called test suites.

It's good practice, with Plone, to prefix all robot test suite files with `robot_`. This makes it easier to both exclude the robot tests (which are usually very time consuming) from test runs or run only the robot tests.

Write a simple robot tests suite `my.product/src/my/product/tests/robot_hello.txt`:

```
*** Settings ***

Library  Selenium2Library  timeout=10  implicit_wait=0.5

Suite Setup  Start browser
```

```
Suite Teardown  Close All Browsers

*** Variables ***

${BROWSER} =  Firefox

*** Test Cases ***

Hello World
    [Tags]  hello
    Go to  http://localhost:55001/plone/hello-world
    Page should contain  Hello World!

*** Keywords ***

Start browser
    Open browser  http://localhost:55001/plone/  browser=${BROWSER}
```

Note: Defining browser for `Open browser` keyword as a variable makes it easy to run the test later with different browser.

1.2.5 Register the suite for zope.testrunner

To be able to run Robot Framework test suite with zope.testrunner and on top of our add-ons functional test fixture, we need to

1. wrap the test suite into properly named Python unittest test suite
2. assign our functional test layer for all the test cases.

We do this all by simply writing `my.product/src/my/product/tests/test_robot.py`:

```
from my.product.testing import MY_PRODUCT_FUNCTIONAL_TESTING
from plone.testing import layered
import robotsuite
import unittest


def test_suite():
    suite = unittest.TestSuite()
    suite.addTests([
        layered(robotsuite.RobotTestSuite("robot_hello_world.txt"),
                layer=MY_PRODUCT_FUNCTIONAL_TESTING)
    ])
    return suite
```

Note that `test_`-prefix in the filename of `test_robot.py` is required for **zope.testunner** to find the test suite.

1.2.6 List and filter tests

Run `bin/test` (**zope.testrunner**) with `--list-tests`-argument to see that our test is registered correctly:

```
$ bin/test --list-tests
Listing my.product.testing.MyproductLayer:Functional tests:
  Hello_World (robot_hello_world.txt) #hello
```

```
Listing my.product.testing.MyproductLayer:Integration tests:
  ...
```

Experiment with `-t`-argument to filter testrunner to find only our robot test:

```
$ bin/test -t robot_ --list-tests
Listing my.product.testing.MyproductLayer:Functional tests:
  Hello_World (robot_hello_world.txt) #hello
```

or everything else:

```
$ bin/test -t \!robot_ --list-tests
Listing my.product.testing.MyproductLayer:Integration tests:
  ...
```

We can also filter robot tests with tags:

```
$ bin/test -t \#hello --list-tests
Listing my.product.testing.MyproductLayer:Functional tests:
  Hello_World (robot_hello_world.txt) #hello
```

1.2.7 Run (failing) test

After the test has been written and registered, it can be run normally with `bin/test`.

The run will fail, because the test describes an unimplemented feature:

```
$ bin/test -t robot_

Running my.product.testing.MyproductLayer:Functional tests:
  Set up plone.testing.zca.LayerCleanup in 0.000 seconds.
  Set up plone.testing.z2.Startup in 0.217 seconds.
  Set up plone.app.testing.layers.PloneFixture in 7.643 seconds.
  Set up my.product.testing.MyproductLayer in 0.026 seconds.
  Set up plone.testing.z2.ZServer in 0.503 seconds.
  Set up my.product.testing.MyproductLayer:Functional in 0.000 seconds.
  Running:
    1/1 (100.0%)
==============================================================================
Robot Hello World
==============================================================================
Hello World                                                           | FAIL |
Page should have contained text 'Hello World!' but did not
------------------------------------------------------------------------------
Robot Hello World                                                     | FAIL |
1 critical test, 0 passed, 1 failed
1 test total, 0 passed, 1 failed
==============================================================================
Output:  /.../my.product/parts/test/robot_hello_world/Hello_World/output.xml
Log:     /.../my.product/parts/test/robot_hello_world/Hello_World/log.html
Report:  /.../my.product/parts/test/robot_hello_world/Hello_World/report.html



Failure in test Hello World (robot_hello_world.txt) #hello
Traceback (most recent call last):
  File "/.../unittest2-0.5.1-py2.7.egg/unittest2/case.py", line 340, in run
    testMethod()
```

```
  File "/.../eggs/robotsuite-1.0.2-py2.7.egg/robotsuite/__init__.py", line 317, in runTest
    assert last_status == 'PASS', last_message
AssertionError: Page should have contained text 'Hello World!' but did not


  Ran 1 tests with 1 failures and 0 errors in 3.632 seconds.
Tearing down left over layers:
  Tear down my.product.testing.MyproductLayer:Functional in 0.000 seconds.
  Tear down plone.testing.z2.ZServer in 5.282 seconds.
  Tear down my.product.testing.MyproductLayer in 0.003 seconds.
  Tear down plone.app.testing.layers.PloneFixture in 0.084 seconds.
  Tear down plone.testing.z2.Startup in 0.006 seconds.
  Tear down plone.testing.zca.LayerCleanup in 0.004 seconds.
```

1.2.8 Create an example view

Create view described in the test by registering a template into `my.product/src/my/product/configure.zcml`:

```
<configure
    xmlns="http://namespaces.zope.org/zope"
    xmlns:five="http://namespaces.zope.org/five"
    xmlns:browser="http://namespaces.zope.org/browser"
    xmlns:i18n="http://namespaces.zope.org/i18n"
    xmlns:genericsetup="http://namespaces.zope.org/genericsetup"
    i18n_domain="my.product">

  ...

  <browser:page
      name="hello-world"
      for="Products.CMFCore.interfaces.ISiteRoot"
      template="hello_world.pt"
      permission="zope2.View"
      />

  ...

</configure>
```

And writing the template into `my.product/src/my/product/hello_world.pt`:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
      xmlns:tal="http://xml.zope.org/namespaces/tal"
      xmlns:metal="http://xml.zope.org/namespaces/metal"
      xmlns:i18n="http://xml.zope.org/namespaces/i18n"
      lang="en"
      metal:use-macro="context/main_template/macros/master"
      i18n:domain="plone">
<body>

<metal:content-core fill-slot="content-core">
    <metal:content-core define-macro="content-core">
      <p>Hello World!</p>
    </metal:content-core>
</metal:content-core>

</body>
```

```
</html>
```

1.2.9 Run (passing) test

Re-run the test to see it passing:

```
$ bin/test -t robot_
Running my.product.testing.MyproductLayer:Functional tests:
  Set up plone.testing.zca.LayerCleanup in 0.000 seconds.
  Set up plone.testing.z2.Startup in 0.220 seconds.
  Set up plone.app.testing.layers.PloneFixture in 7.810 seconds.
  Set up my.product.testing.MyproductLayer in 0.027 seconds.
  Set up plone.testing.z2.ZServer in 0.503 seconds.
  Set up my.product.testing.MyproductLayer:Functional in 0.000 seconds.
  Running:

  Ran 1 tests with 0 failures and 0 errors in 2.604 seconds.
Tearing down left over layers:
  Tear down my.product.testing.MyproductLayer:Functional in 0.000 seconds.
  Tear down plone.testing.z2.ZServer in 5.253 seconds.
  Tear down my.product.testing.MyproductLayer in 0.004 seconds.
  Tear down plone.app.testing.layers.PloneFixture in 0.085 seconds.
  Tear down plone.testing.z2.Startup in 0.006 seconds.
  Tear down plone.testing.zca.LayerCleanup in 0.004 seconds.
```

1.2.10 Test reports

Robot Framework generates high quality test reports with screenshots of failing tests as:

`my.product/parts/tests/robot_report.html` Overview of the test results.

`my.product/parts/tests/robot_log.html`: Detailed log for every test with screenshots of failing tests.

1.3 Learn more robot

Robot Framework is a generic and independent test automation framework with its own expandable test syntax, test runner and test reporting tools. Yet, because of its extensibility it's very pleasant to work with.

Robot is all about running test clauses called **keywords**. Every test may contain one more keywords, which are run in serial – usually until first of them fails.

Keywords are defined in **keywords libraries** and as **user keywords**. Keyword libraries may be, for example, Python libraries or XML-RPC-services. User keywords are just composite lists of existing keywords – also user keywords.

Because user keywords can also be composite of other user keywords, they make it possible to write tests in domain-specific language.

1.3.1 Test suite

Robot tests are written in test suites, which are simply plain text files, usually ending with `.txt`.

Note: Advanced robot users may learn from the Robot Framework User Guide how to make hierarchical test suites.

Here's an example test suite:

```
*** Settings ***

Library  Selenium2Library  timeout=10  implicit_wait=0.5
Resources  keywords.txt

Suite Setup  Start browser
Suite Teardown  Close All Browsers

*** Variables ***

${BROWSER} =  firefox

*** Test Cases ***

Hello World
    [Tags]  hello
    Go to  http://localhost:55001/plone/hello-world
    Page should contain  Hello World!

*** Keywords ***

Start browser
    Open browser  http://localhost:55001/plone/  browser=${BROWSER}
```

Each test suite may contain one to three different parts:

`**Settings**`

> Import available keyword libraries or resources (resources are plain text files like test suites, but without test cases) and define possible setup and teardown keywords.

Variables Define available robot variables with their default values.

Test Cases Define runnable tests.

Keywords Define new user keywords.

1.3.2 BDD-style tests

Robot support Gherkin-style tests by removing exact words `given`, `when`, `then` and `and` from the beginning of keyword to find a matching keyword.

For example, a clause:

```
Given I'm logged in as an admin
```

will match to a keyword:

```
I'm logged in as an admin
```

There's a little bit more of BDD-style tests in Robot Framework User Guide.

1.4 Debugging robot tests

1. Slow done Selenium (WebDriver) to make the tests easier to follow:

```
Set Selenium Speed  0.5 seconds
```

2. Pause Selenium (WebDriver) completely to inspect your step:

```
Set Selenium Timeout  600 seconds
Wait For Condition  true
```

3. Write a python keyword into your Python keyword library to drop the Zope server into debugger.

   There's one catch in debugging your code while running Robot Framework tests. Robot eats your standard input and output, which prevents you to just `import pdb; pdb.set_trace()`.

   Instead, you have to add a few more lines to reclaim your I/O at first, and only then let your debugger in:

```
import sys
import pdb
for attr in ('stdin', 'stdout', 'stderr'):
    setattr(sys, attr, getattr(sys, '__%s__' % attr))
pdb.set_trace()
```

CHAPTER

TWO

PRINT THESE

Robot Framework built-in library documentation http://robotframework.googlecode.com/hg/doc/libraries/BuiltIn.html?r=2.7.6

Robot Framework Selenium2Library documentation http://rtomac.github.com/robotframework-selenium2library/doc/Selenium2Library.html

CHAPTER

THREE

READ MORE

How to write good Robot Framework test cases http://code.google.com/p/robotframework/wiki/HowToWriteGoodTestCases

CHAPTER

FOUR

BECOME MASTER

4.1 Speed up your test writing with ACT-server

plone.act comes with a special console script `act_server`, which starts up a Plone site with a given plone.app.testing testing layer set up.

This will save time when writing new robot tests, because you can try out your unfinished test over and over again without the usual time consuming setup/teardown of testing layers between every test.

Install `act_server` with support for the developed product with a buildout part:

```
[buildout]
...
parts += act_server
versions = versions

extensions = mr.developer
sources = sources
auto-checkout = plone.act

[sources]
plone.act = git git://github.com/plone/plone.act

[versions]
plone.app.testing = 4.2.2

[act_server]
recipe = zc.recipe.egg
eggs =
    plone.act
    my.product
```

After buildout, start `act_server` with:

```
$ bin/act_server my.product.testing.MY_PRODUCT_FUNCTIONAL_TESTING
```

And run tests with `pybot` and `act_server` test isolation support with:

```
$ bin/pybot --listener plone.act.server.ZODB src/my/product/tests/robot_tests.txt
```

4.2 Speed up your BDD Given-clauses via a remote library

BDD-style tests begin with one or more *Given*-clauses that should setup the test environment for the actual tests-clauses (*When* and *Then*).

Because Given-clauses are not really part of the actual test, it is not necessary to run them through Selenium (using Selenium2Library), but it would be faster to write custon Python keywords for them.

plone.act includes an example, how to a robot remote library, which could be called to interact with the site without Selenium.

The base implementation is provided at:

https://github.com/plone/plone.act/blob/master/src/plone/act/remote.py

An example integration into testing layer is provided at:

https://github.com/plone/plone.act/blob/master/src/plone/act/testing.py#L43

An example test suite using the library is provided at:

https://github.com/plone/plone.act/blob/master/src/plone/act/tests/test_robot.py#L13

https://github.com/plone/plone.act/blob/master/src/plone/act/tests/robot_remote.txt