Table of Contents

Introduction

Robot Framework is a Python-based, extensible keyword-driven test automation framework for end-to-end acceptance testing and acceptance-test-driven development (ATDD). It can be used for testing distributed, heterogeneous applications, where verification requires touching several technologies and interfaces.

Why Robot Framework

- Enables easy-to-use tabular syntax for creating test cases in a uniform way.
- Provides ability to create reusable higher-level keywords from the existing keywords.
- Provides easy-to-read result reports and logs in HTML format.
- Is platform and application independent.
- Provides a simple library API for creating customized test libraries which can be implemented natively with either Python or Java.
- Provides a command line interface and XML based output files for integration into existing build infrastructure (continuous integration systems).
- Provides support for Selenium for web testing, Java GUI testing, running processes, Telnet, SSH, and so on.
- Supports creating data-driven test cases.
- Has built-in support for variables, practical particularly for testing in different environments.
- Provides tagging to categorize and select test cases to be executed.
- Enables easy integration with source control: test suites are just files and directories that can be versioned with the production code.
- Provides test-case and test-suite -level setup and teardown.
- The modular architecture supports creating tests even for applications with several diverse interfaces.

High-level architecture

Robot Framework is a generic, application and technology independent framework. It has a highly modular architecture illustrated in the diagram below.

Robot Framework architecture

The test data is in simple, easy-to-edit tabular format. When Robot Framework is started, it processes the test data, executes test cases and generates logs and reports. The core framework does not know anything about the target under test, and the interaction with it is handled by test libraries. Libraries can either use application interfaces directly or use lower level test tools as drivers.

Robot Framework Quickstart Guide

This is a guide made to get you up and running quickly! It is not by any means an exhaustive list of the functionality and potential of Robot Framework. It's purpose is to show you how to do the basic stuff in a basic way. This doesn't mean you can't do any useful work with it, quite the opposite. With only the knowledge from this part of the guide, you can create powerful testing suites!

Installation

A precondition for Robot Framework is a recent version of python. For that, Python 2.6 from the EPEL repository was selected and the provided packages are built against it. Then, you must install the robot_testing repository residing at yum.gridctb.uoa.gr and after that, you can install the packages required.

```
yum install python26
wget http://yum.gridctb.uoa.gr/repository/robot_testing.repo -O /etc/yum.repos.d/robot_testing.re
yum install wxPython-common-gtk2-unicode wxPython2.8-gtk2-unicode
yum install robotframework pycrypto paramiko SSHLibrary robotframework-ride
```

Stuff You Need to Know

Robot Framework is a Python-based, extensible keyword-driven test automation framework. It uses text or html table files, describing the testing steps. These files are comprised of keywords and variables. A keyword is a synonym of a "function" doing "something". A variable is well...a variable! Tests are created from testsuites. Testsuites are created from tables. The available tables are: (if some things will sound unknown,don't worry,they shall be explained shortly)

- Settings
  - Used to include libraries, among other things. Can have a Setup and/or a Teardown. Can have Documentation. Can have tags. Can have comments.
- Variable
  - Used to declare variables.
- Keyword
  - Used to declare user keywords. Can have Documentation. Can have comments.
- Test Case
  - Used to create the actual test. Can have a Setup and/or a Teardown. Can have Documentation. Can have tags. Can have comments.

A Setup is a set of things to happen BEFORE the test is executed -e.g.: set up the environment- . A Teardown is a set of thing to be done AFTER the test is executed -e.g.: clean up- . Note that Teardowns are ALWAYS executed, regardless of errors during the test or during the Teardown itself. Also note that Setups/Teardowns must be a single keyword, thus in case of a complex situation, a user level keyword containing all the keywords you want to execute must be created.

Setups/Teardowns in the Settings table,will be executed once before and after the testsuite. Setups/Teardowns in the Test Case table, will be execute once before and agter the test case.

Tags are used as synonyms. For example, a test case name may be too big or non descriptive. Its tag may have those properties. A test suite or a test case can have multiple tags.

Documentation is used to give short descriptions regarding the test element being documented.

A user keyword is almost like a test case. For example, if a certain part of the test is executed multiple times,it could be written as a user keyword. User keywords could be imagined as "functions" in the context of

imperative programming languages. i.e.: they reduce the clutter, provide better maintainability, they provide better understanding of the algorithm etc.

Comments start with # and are used as in any other context. They do not have any other effect.

Keywords may or may not take arguments and may or may not return a value. Arguments can be either set explicitly or be assigned first to variables. Return values may be stored in variables. Variables can be either a simple variable (string,number etc) or a list containing multiple variables. Generally, you will use list variables a lot less frequently.

Keywords come from libraries. A library is a python module coming with Robot Framework or created by you. The module's functions, are the library's keywords and vice versa.

The libraries that come with Robot Framework are: (with their respective original documentation)

- BuiltIn http://robotframework.googlecode.com/hg/doc/libraries/BuiltIn.html
- OperatingSystem http://robotframework.googlecode.com/hg/doc/libraries/OperatingSystem.html
- Telnet http://robotframework.googlecode.com/hg/doc/libraries/Telnet.html
- Collections http://robotframework.googlecode.com/hg/doc/libraries/Collections.html
- String http://robotframework.googlecode.com/hg/doc/libraries/String.html
- Dialogs http://robotframework.googlecode.com/hg/doc/libraries/Dialogs.html
- Screenshot http://robotframework.googlecode.com/hg/doc/libraries/Screenshot.html
- SSH http://robotframework-sshlibrary.googlecode.com/svn/tags/robotframework-sshlibrary-1.0/doc/SSHLibrary.htr

The Collections library is providing methods for handling Python lists and dictionaries. The rest of the library names are pretty much self explanatory, regarding the keywords/methods/functions they provide.

Any library besides BuiltIn, must be explicitly included. This is done in the Settings table (more on that in a bit).

Robot framework creates html and xml reports uppon completion. The html reports log which tests passed or failed and how/why. The xml report contains the same data in xml format, to be easily used for post processing.

If you want detailed information in your reports,set the command line argument

```
-L TRACE
```

. If you don't want detailed information,set the command line argument

```
-L WARN
```

.

Tags main use is to manipulate individual test cases. In this fashion, test cases can be included or excluded from a testsuite execution, or their criticality can be set. (by default all tests are "critical",which means that their failure results in the testsuite's failure) I will show you how to do this later in this part of the guide.

You can use any text or html table editor you like for your test cases. In the start, a simple text editor would suffice. Once your testing needs get bigger or you need to experiment further,the RIDE editor is the proposed solution for test case editing. RIDE is a wysiwyg editor providing integration with Robot Framework,syntax highlighting and able to output test cases in simple text or html table format.

If you want to create your own library, there are some very basic things you need to remember:

- You must know python
- Any method in the python module -NOT starting with underscore(s) that is- can be used as a keyword.
- The method for setting and reporting failure is through python exceptions. So,supposing that your python method -a.k.a. library keyword- wants to check if a file contains a certain word,it could raise an exception if the words wasn't found. Alternatively, it could return a True/False value, or anything dictated by your app logic and your testing needs.
- Your custom library can be included like any other library. Most probably, you will/should enter its absolute path.

Examples

All the tests in this section should saved in a simple text file -lets say test.txt- . They can be executed by running:

```
pybot test.txt
```

Once the test is finished, the reports will be available in the current working directory.

Simple Test

This is a simple test,containing only a Test Case table. The cell under "*Test Case*" contains the name of the test case. The cell under "*Action*" contains the action to be taken, described by a keyword. The cells under "*Argument*" cells can be anything else (such as comments or arguments or variables etc).

This test checks whether its second string argument "world" is contained within its first string argument "Hello, world!"

```
| *Test Case* |  *Action*              |           *Argument*               |    *Argument*
| Simple Test | Should Contain         |   Hello, world!                    | world
```

Include Library

This test shows how to include an extra library. In this case we include the OperatingSystem library.

This test creates a file and appends a string to it.

```
| *Setting* |     *Value*           |
| Library   | OperatingSystem       |

| *Test Case*   |  *Action*        |           *Argument*          |    *Argument*    |
| Simple Test 2 | [Documentation]  |   Create a temp file          |                  |
|               | Create File      |   /tmp/robot_test.txt         |                  |
|               | Append To File   |   /tmp/robot_test.txt         |  2b || ! 2b      |
```

Set a tag

This is the same as the simple test.

This test case demonstrates how a tag can be set.

```
| *Test Case* |  *Action*              |           *Argument*               |    *Argument*
|             | [Tags]                 |   check_str                        |
| Simple Test | Should Contain         |   Hello, world!                    | world
```

Use a variable

This is the same as the simple test.

This test case illustrates how to set and use a variable.

```
| *Variable* | *Value*  |
| ${str}     |  world   |

| *Test Case* |  *Action*             |           *Argument*               |    *Argument*
| Simple Test | Should Contain        |   Hello, world!                    | ${str}
```

Store a return value to a variable

This test illustrates how to store a keyword return value in a variable.

This test case reads the time and then it logs it. The value will be present in the report. The value will be present in the report, because of the "Log" keyword.

```
| *Test Case* |  *Action*             |           *Argument*               |
| Time        | ${time}=              |   Get Time                         |
|             | Log                   | ${time}                            |
```

Store multiple return values to variables

This test illustrates how to store a keyword's return values to seperate variables.

This test case read the time in hour:minute format and then it logs it. The value will be be present in the report, because of the "Log" keyword.

```
| *Test Case* |  *Action*             |           *Argument*               |           *Argument
| Time        | ${hour}               | ${minute}                          |   Get Time
|             | Log                   | ${hour}:${minute}                  |
```

Use Comments

This test is the same as the simple test.

It shows the usage of comments

```
| *Test Case* |  *Action*             |           *Argument*               |    *Argument*
| Simple Test | Should Contain        |   Hello, world!                    | world
```

User Keywords

This is a test demonstrating how to define and use User Keywords.

In this test suite, we first import the operating system library, in order to use the "Count Files In Directory" keyword. This keyword's purpose is self explanatory. Then we create two variables,${path1} and ${path2} which hold two paths in the local filesystem. Under the Variable Table, there is the Keyword Table. There, we define our own keyword. What it does is that it counts the number of files in the given directory, then reads the time and then logs a message with the results. Note how you can describe that a user keyword takes an argument. A user keyword might take none, one, or more arguments. How we store the return values and how we log a message is already shown in previous examples. Finally, in the Test Case Table, we define some documentation and a tag for our test case. Then we call the user keyword, with the one argument it expects,

which is define earlier in the Variables Table. In the end,two messages will be logged in the test's report, saying how many files the paths contained and what time was the measurement taken.

```
| *Setting* |     *Value*           |
| Library   | OperatingSystem       |

| *Variable*  |  *Value*   |
| ${path1}    |  /tmp      |
| ${path2}    |  /opt      |

| *Keyword* |  *Action*    |  *Argument*                                 |  *Argument*        |
| My_KW     |  [Arguments] | ${path}                                     |                    |
|           |  ${count}=   | Count Files In Directory                    | ${path}            |
|           |  ${time}=    | Get Time                                    |                    |
|           |  Log         | At ${time} path ${path} had ${count} items  |                    |

| *Test Case*   |  *Action*        |           *Argument*         |    *Argument*    |
| User KW       | [Documentation]  |   Use a user keyword         |                  |
|               | [Tags]           |  user_kw                     |                  |
|               | My KW            |  ${path1}                    |                  |
|               | My KW            |  ${path2}                    |                  |
```

Setup and Teardown

This is a test demonstrating how to use a test case setup and teardown. In this, we interchange the contents of two files, with the use of a middle file.

In this test suite, first we import the operating system library. Then, we create a user keyword which creates three files. The first two file, have the contents given in the second argument. The third file is left empty. In the test case, we define the setup to the keyword we created, since a single keyword wouldn't suffice -remember, Setups/Teardowns must be a single keyword!- . Also, we define teardown, in which we delete the middle file since we no longer need it at the end of the test. The test case uses a simple algorithm to change the contents of the two files. It first copies the first to the middle, then the second to the first and finally the middle to the second.The contents of the files /tmp/temp_file_one.txt and /tmp/temp_file_two.txt should now be swapped.

```
| *Setting* |     *Value*           |
| Library   | OperatingSystem       |

| *Keyword*           |  *Action*           |  *Argument*                       |  *Argument*
| Create Three Files  |  [Documentation]    | Creates three files under /tmp    |
|                     |  Create File        | /tmp/temp_file_one.txt            | File One
|                     |  Create File        | /tmp/temp_file_two.txt            | File Two
|                     |  Create File        | /tmp/temp_file_middle.txt         |

| *Test Case*   |  *Action*        |           *Argument*                |    *Argument*
| Set_n_Tear    | [Documentation]  |  Change the contents of two files   |
|               | [Setup]          |  Create Three Files                 |
|               | [Teardown]       |  Remove File                        | /tmp/temp_file_middle
|               | Copy File        |  /tmp/temp_file_one.txt             | /tmp/temp_file_middle
|               | Copy File        |  /tmp/temp_file_two.txt             | /tmp/temp_file_one.tx
|               | Copy File        |  /tmp/temp_file_middle.txt          | /tmp/temp_file_two.tx
```

User Created Library

This is a python module containing an exception (_error) class and a method. The method checks whether a string is contained within a file or not. Save this file in /tmp/test_mod.py

```
class _error(Exception):
        def __init__(self,string):
                self.string = string
```

```
        def __str__(self):
                return str(self.string)

def file_should_contain(file_path,search_string):
        file_path=file_path.encode('ascii')
        search_string=search_string.encode('ascii')

        file_as_string=open(file_path).read()
        if search_string in file_as_string:
                return True

       raise _error("The string " + search_string +  " wasn't found in file " + file_path)
```

Then,create the following test case. The following test case is designed to fail (in the off chance that your dmesg file indeed contains the string "foobar", then it will of course succeed!)

```
| *Setting* |     *Value*           |
| Library   | /tmp/test_mod.py      |

| *Test Case*   |  *Action*              |           *Argument*                |    *Argument*
| User Lib      |  [Documentation]       | This test uses a user library       |
|               |  File Should Contain   | /var/log/dmesg                      |  error
|               |  File Should Contain   | /var/log/dmesg                      |  foobar
```

All this should be pretty self explanatory be now. We have created a test case which checks if the file given in the first argument, contains the string given in the second argument. Following the python code from above, the first test should probably succeed (it is rather usual for the string "error" to be found in dmesg) and the second should (probably) fail.

One thing you should notice is that the keyword name is case insensitive and space insensitive. This is to allow more human readable keywords.

Of course there is much more to library building with Robot Framework, but this would get you started!

An EMI Related Example

This is a python module containing an exception class (_error) and a method. The method provided is "create_proxy", which acts as a keyword-wrapper for the voms-proxy-init command. It utilizes the subprocess module for spawning and tracing new processes (its documentation can be found here: http://docs.python.org/library/subprocess.html), and the shlex module for simple lexical analysis (its documentation can be found here: http://docs.python.org/library/shlex.html). The method accepts two parameters, the user's password and the vo used for the voms extension. Save this file in /tmp/test_mod.py

```
import subprocess , shlex

class _error(Exception):
        def __init__(self,string):
                self.string = string
        def __str__(self):
                return str(self.string)

def create_proxy(password,vo):
        '''
                Description:    Create a user proxy.
                Arguments:     password,vo for the voms extention.
                Returns:         nothing.
        '''

        com="/usr/bin/voms-proxy-init -pwstdin --voms %s" % vo
        args = shlex.split(com.encode('ascii'))
        p = subprocess.Popen( args , shell=False , stderr=subprocess.STDOUT , stdout=subprocess.P
```

```
        p.communicate(input=password)

        retVal=p.wait()

        if retVal != 0 :
                if retVal == 1 :
                        raise _error("Proxy creation failed.Most probably wrong Virtual Organisat
                elif retVal == 3 :
                        raise _error("Proxy creation failed.Most probably the password provided w
                else :
                        raise _error("Proxy creation failed.Reason: Unknown.")
```

Then,create the following test case. The following test case doesn't do anything particularly useful, beyond creating a temporary proxy.

```
| *Setting* |     *Value*           |
| Library   | /tmp/test_mod.py      |

| *Test Case*   |  *Action*              |           *Argument*                |    *Argument*
| User Lib      |  [Documentation]       | This test creates a proxy           |
|               |  Create Proxy          | p4sSw0rD                            | dteam
```

Cream Testing: A full blown test suite example of Robot Framework

At NKUA we are the developers and maintainers of the new CREAM functionality test suite. It is developed for Robot Framework. The keywords used, beyond the ones provided by the framework itself, are developed by us in an external library. If you want to expand your experiments with Robot Framework or gain a better understanding of how it works, you can find more information here: https://twiki.cern.ch/twiki/bin/view/EMI/CREAMRobotFuncTests. Also, the package available at the robot_testing repo "cream_test" will install the aforementioned test suite. It depends on lcg-utils, lfc and glite-ce-cream-cli. It is designed to be deployed in an emi-ui, but it can be installed in any relevant environment.

Useful Command Line Options

These are some basic useful command line options. See the appendix for a full list.

Exclude Test Case by Tag

A test case can be excluded from execution, using the -e argument.

```
pybot -e tag_name test.txt
```

Include Test Case by Tag

The test cases to be executed can be selected, using the -i argument.

```
pybot -i tag_name test.txt
```

Set Criticality by Tag

Whether a test case is critical to the test suite or not, can be set by using the --critical or --noncritical argument.

```
pybot --critical tag_name text.txt
```

Set Log Level

The log level of a test suite can be set, using the -L argument. Possible values are FAIL, WARN, INFO, DEBUG, TRACE

```
pybot -L TRACE test.txt
```

Set Terminal Screen Width

You can set the length of the output shown by Robot Framework by using the -W argument. This is used in order for the output to fit your terminal screen, because Robot Framework's default is rather small. The $COLUMNS variable is rather useful in this situation.

```
pybot -W 125 test.txt
```

Advanced Guide

The advanced guide for Robot Framework can be found here:
https://twiki.cern.ch/twiki/bin/view/EMI/RobotFrameworkAdvancedGuide

-- DimosthenesFioretos - 16-Oct-2011

This topic: EMI > RobotFrameworkQuickstartGuide
Topic revision: r2 - 2011-10-17 - DimosthenesFioretosExCern